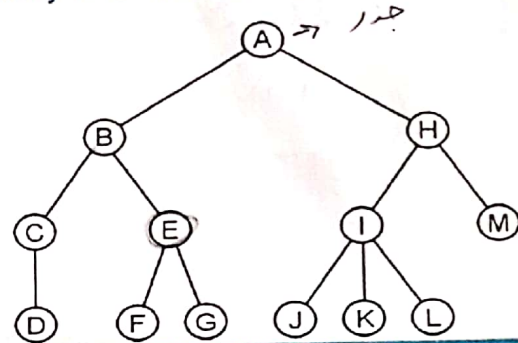


Trees

A rooted tree data structure stores information in *nodes*

– Similar to linked lists:

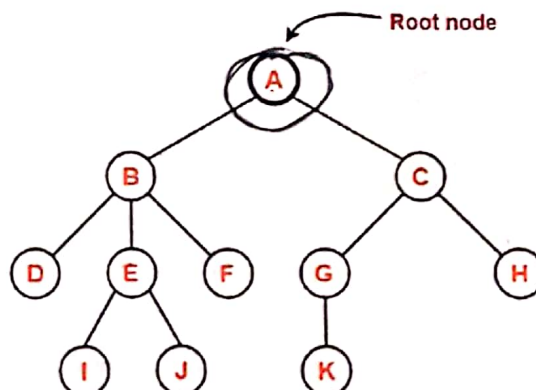
- There is a first node, or *root*
- Each node has variable number of references to successors (children)
- Each node, other than the root, has exactly one node as its predecessor (or parent)



Terminology

Root is the topmost node of the tree.

There is only one root per tree and one path from the root node to any node.



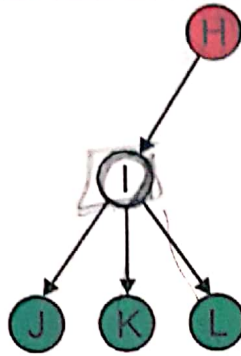
Terminology

All nodes will have zero or more child nodes or *children*

- I has three children: J, K and L

For all nodes other than the root node, there is one *parent* node

- H is the parent of I

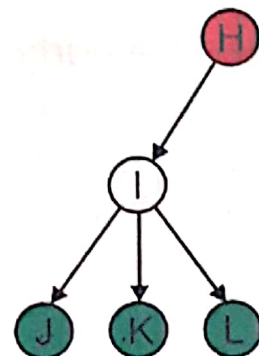


Terminology

The *degree* of a node is defined as the number of its children: $\deg(I) = 3$

Nodes with the same parent are *siblings*

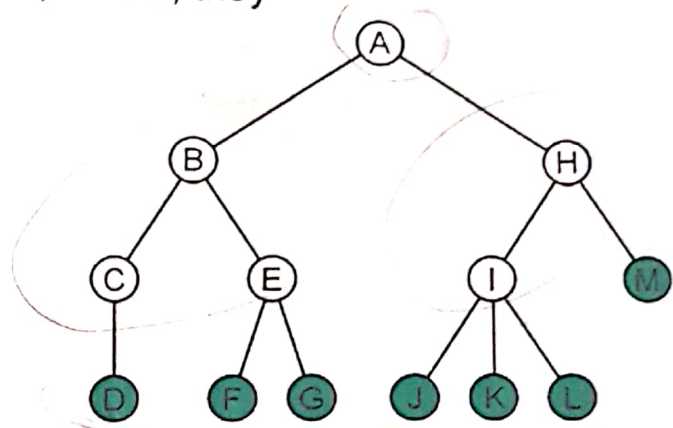
- J, K, and L are siblings



Terminology

Nodes with degree zero are also called *leaf nodes*

All other nodes are said to be *internal nodes*, that is, they are internal to the tree

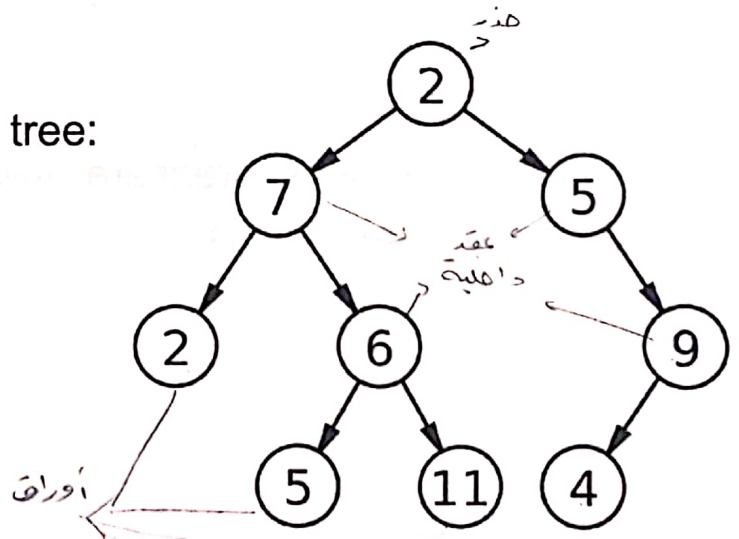


Example

Trees:

Here's a purty picture of a tree:

- 2 is the **root**
- 2, 5, 11, and 4 are **leaves**
- 7, 5, 6, and 9 are **interior nodes**



Terminology

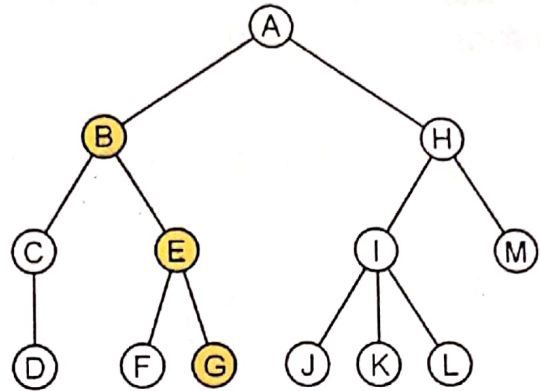
المسار هو سلسلة من العقد
A path is a sequence of nodes

$(a_0, a_1, \dots, a_n)$

حيث a_{k+1} هو ابن a_k
where a_{k+1} is a child of a_k is

طول هذا المسار هو n
The length of this path is n

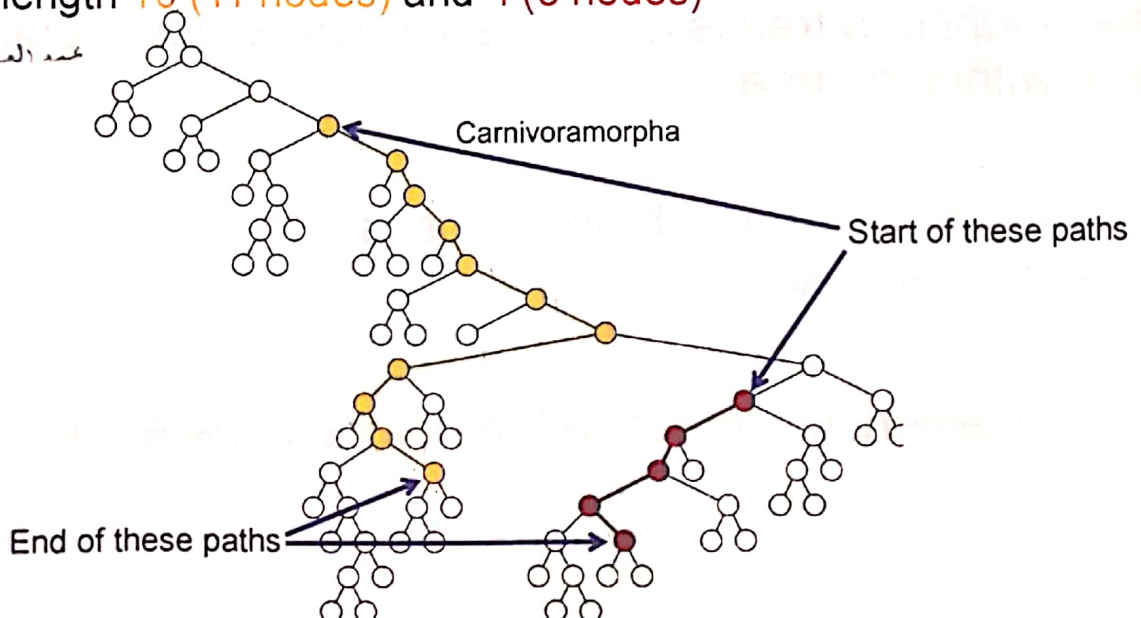
المسار (B, E, G) له طول 2
E.g., the path (B, E, G) has length 2



Terminology

مسارات بطول 10 (11 nodes) و 4 (5 nodes)
Paths of length 10 (11 nodes) and 4 (5 nodes)

طول المسار = عدد العقد - 1
length = number of nodes - 1

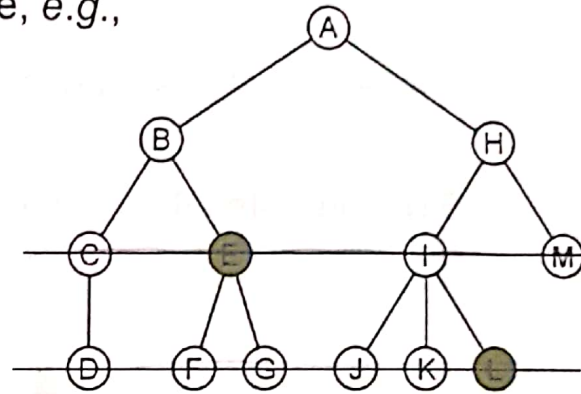


Terminology

For each node in a tree, there exists a unique path from the root node to that node

The length of this path is the *depth* of the node, e.g.,

- E has depth 2
- L has depth 3



Terminology

The *height* of a tree is defined as the maximum depth of any node within the tree

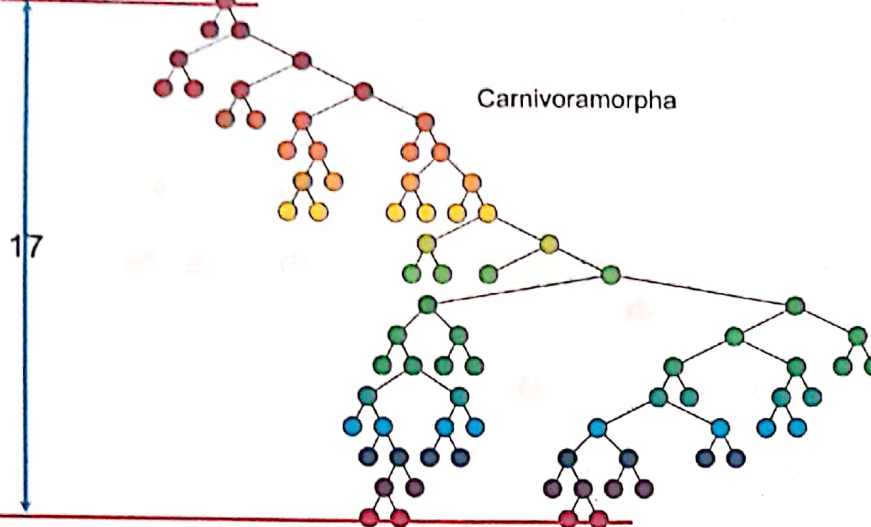
The height of a tree with one node is 0

- Just the root node

For convenience, we define the height of the empty tree to be -1

Terminology

The height of this tree is 17



Terminology

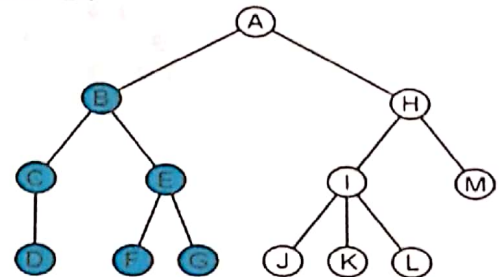


إذا كان مسار من العقدة إلى العقدة
If a path exists from node a to node b :
 - a is an *ancestor* of b
 - b is a *descendent* of a

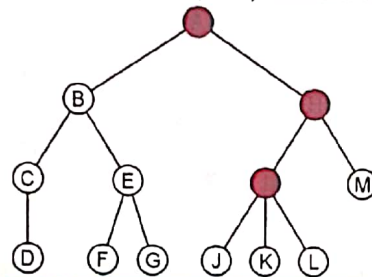
Thus, a node is both an ancestor and a descendant of itself
 The root node is an ancestor of all nodes

Terminology

The **descendants** of node B are B, C, D, E, F, and G:



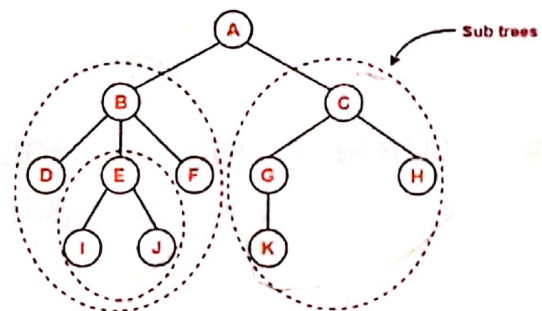
The **ancestors** of node I are I, H, and A:



Terminology

In a tree, each child from a node forms a **subtree** recursively.

Every child node forms a subtree on its parent node.



Example: XHTML

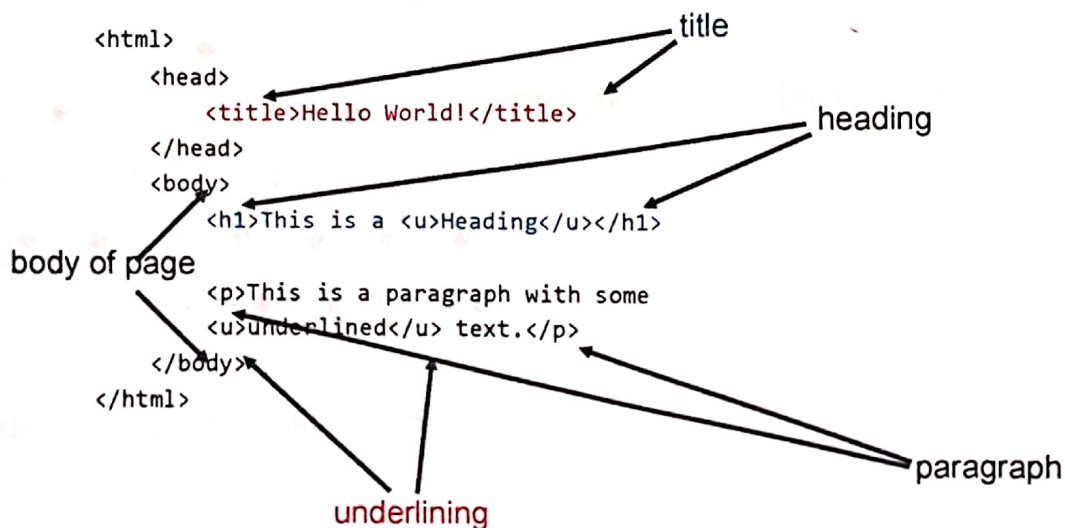
Consider the following XHTML document

```
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>This is a <u>Heading</u></h1>

    <p>This is a paragraph with some
    <u>underlined</u> text.</p>
  </body>
</html>
```

Example: XHTML

Consider the following XHTML document



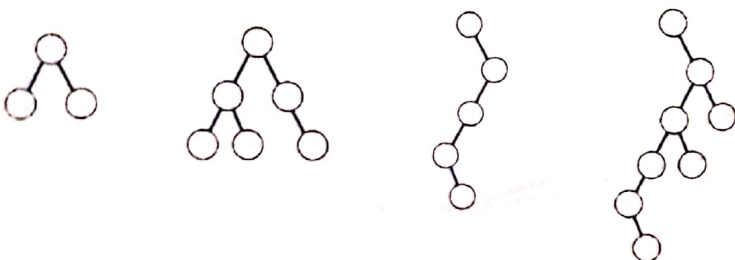
The nested tags define a tree rooted at the HTML tag

```

graph TD
    html --> head
    html --> body
    head --> title
    title --> HW["Hello World!"]
    body --> h1
    body --> p
    h1 --> T1["This is a "]
    h1 --> u1[u]
    u1 --> H["Heading"]
    p --> T2["This is a paragraph with "]
    p --> u2[u]
    p --> T3[" text."]
    u2 --> U["underlined"]
  
```

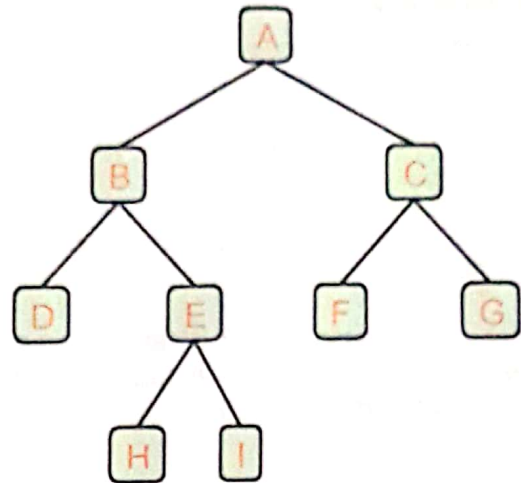
Binary Tree

Every node has degree up to 2.

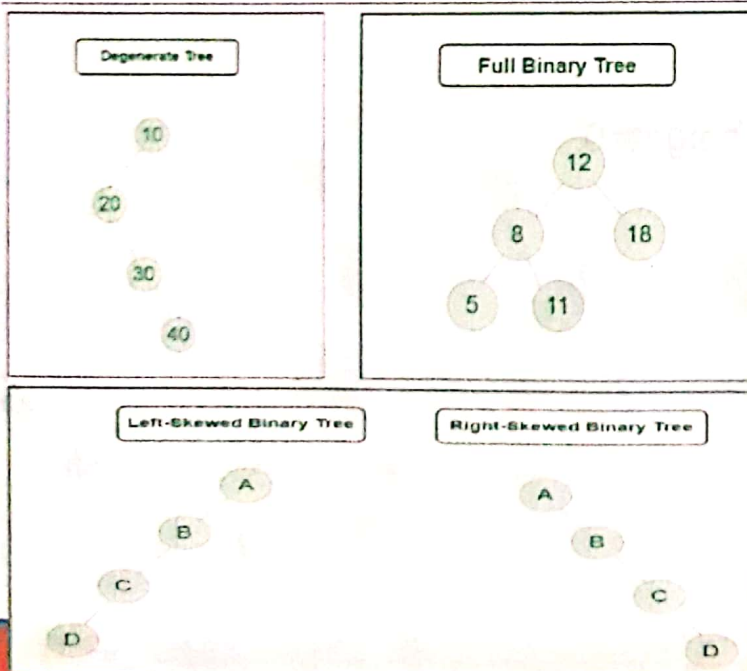


Binary Tree

- We call the children of an internal node **left child** and **right child**
- Applications:
 - arithmetic expressions
 - decision processes
 - searching



Types of Binary Tree based on the number of children



Following are the types of Binary Tree based on the number of children:

1. **Full Binary Tree**: A Binary Tree is a full binary tree if every node has 0 or 2 children

2. **Degenerate Binary Tree**: A Tree, where every internal node has one child.

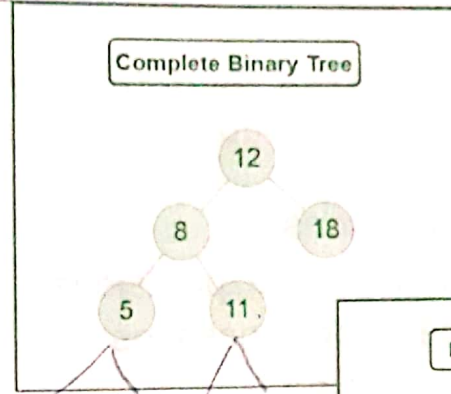
3. **Skewed Binary Trees**: A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes.

Types of Binary Tree On the basis of the completion of levels

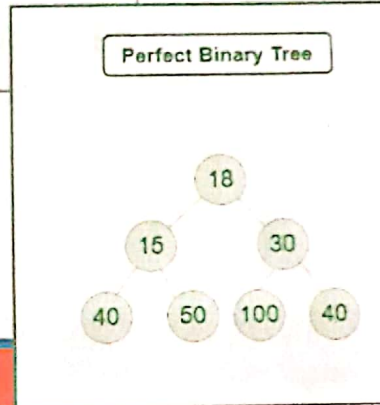


1. Complete Binary Tree: A complete binary tree is just like a full binary tree, but with two major differences:

- Every level except the last level must be completely filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



2. Perfect Binary Tree: A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.



Full
وليد

Some Numbers



For binary tree of height h :

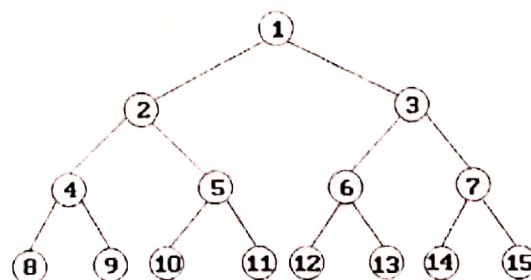
• max # of tree nodes: $2^{h+1}-1$ root height=0

max # of tree leaves: 2^h

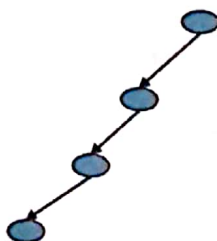
• min # of tree leaves: 1

• min # of tree nodes: $h+1$

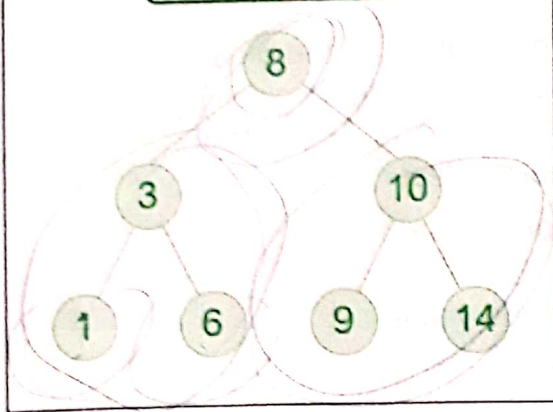
• max # of nodes on level i : $2^i, i \geq 0$.



$h=3$



Binary Search Tree

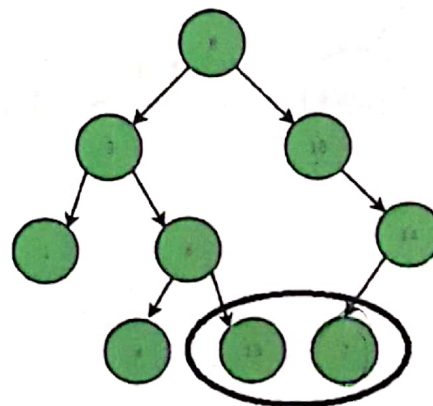
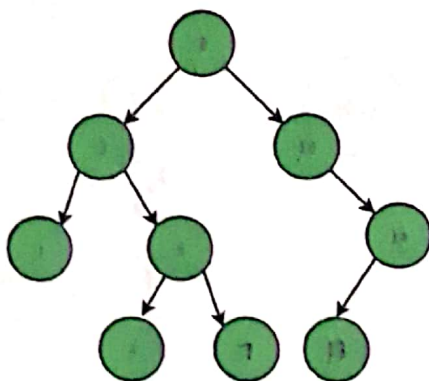


Binary Search Tree

Binary Search Tree is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Example of Binary Search Tree



Insertion in Binary Search Tree (BST)

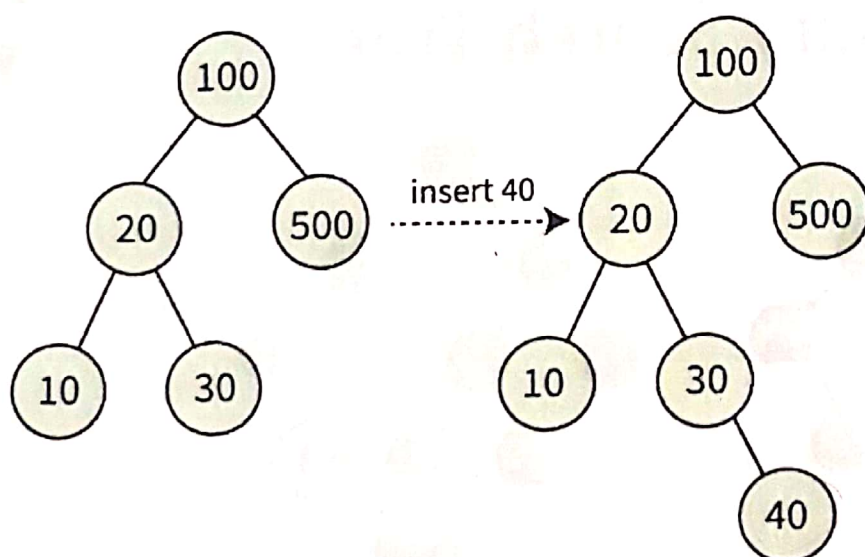
How to Insert a value in a Binary Search Tree?

A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree:

- Initialize the current node (say, **currNode** or **node**) with root node.
- Compare the **key** with the current node.
- **Move left** if the **key** is less than or equal to the current node value.
- **Move right** if the **key** is greater than current node value.
- Repeat steps 2 and 3 until you reach a leaf node.
- Attach the **new key** as a left or right child based on the comparison with the leaf node's value.

Binary tree ليست بالشجرة بكماء Binary search tree

لكنه ليس صحيح



Insertion in Binary Search Tree (BST)

Time Complexity OF Insertion in (BST)

- The worst-case time complexity of insert operations is $O(h)$ where h is the height of the Binary Search Tree.
- In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of insertion operation may become $O(n)$.

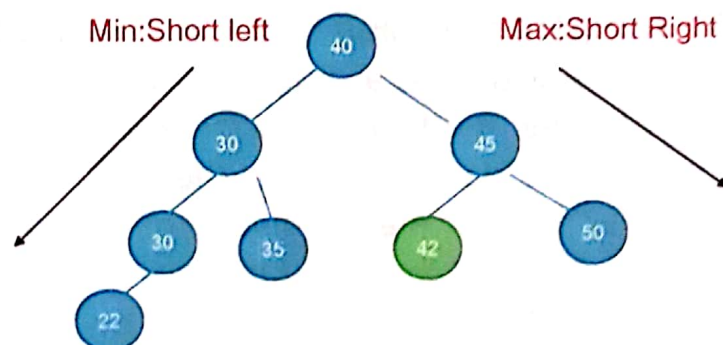
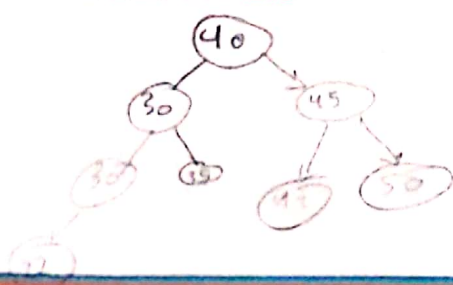
Example-1-

Form Binary search tree from these numbers:

40,45,30,35,30,22,50

Insert 42

Search 35

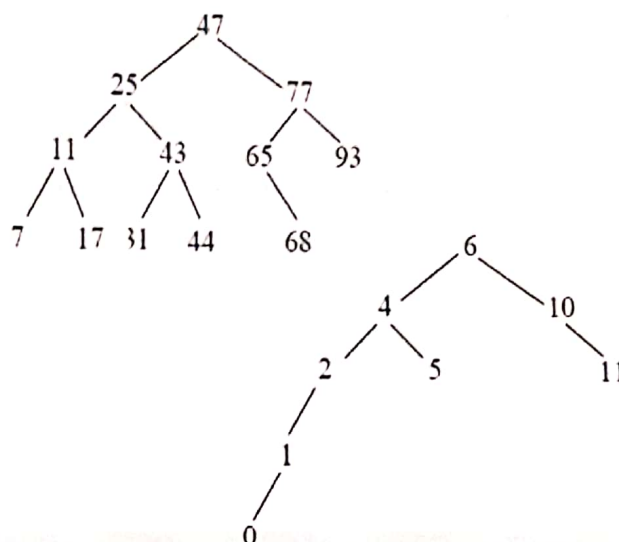




Example-2-

Form Binary search tree from these numbers:

- 47-25-77-11-43-93-65-31-17-7-44-68
- 6, 4, 5, 10, 2, 1, 0, 11



Search in Binary Search Tree (BST)

How to search for the number X in a Binary Search Tree?

We start at the root. Then:

- We compare the value to be searched with the value of the root.
 - If it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
- Repeat the above step till no more traversal is possible
- If at any iteration, key is found, return True. Else False.

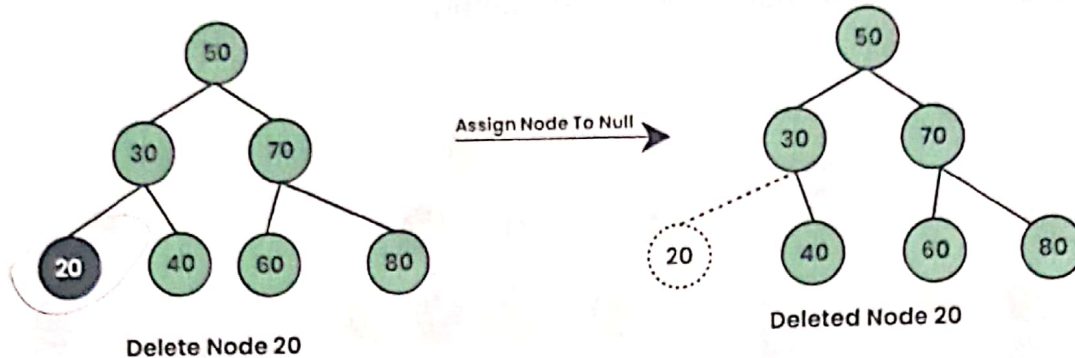
Time complexity: $O(h)$, where h is the height of the BST.



Deletion in Binary Search Tree (BST)

Case 1. Delete a Leaf Node in BST

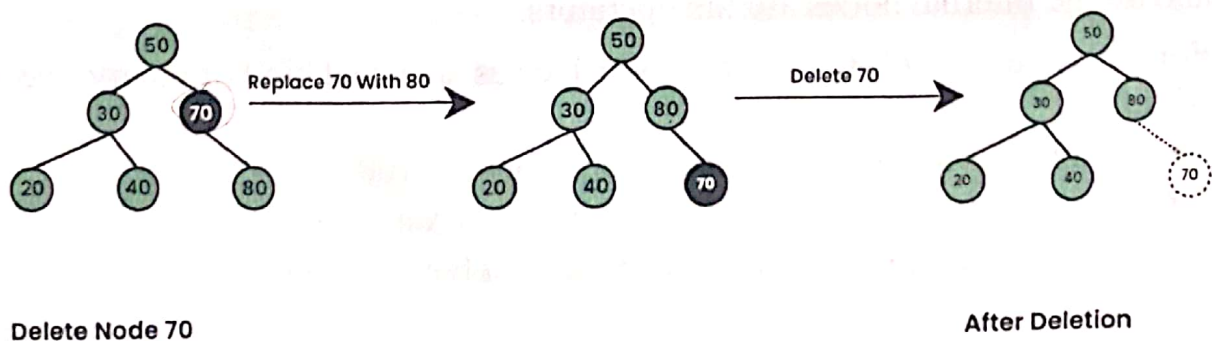
If the node is a leaf, it can be deleted immediately



Deletion in Binary Search Tree (BST)

Case 2. Delete a Node with Single Child in BST

Deleting a single child node is also simple in BST. Copy the child to the node and delete the node.

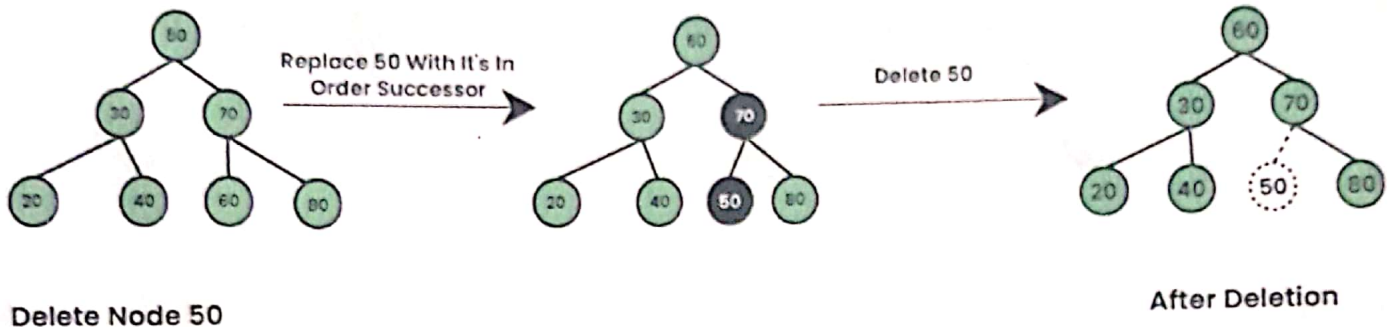




Deletion in Binary Search Tree (BST)

Case 3. Delete a Node with Both Children in BST

The general strategy is to replace the data of this node with the smallest data of the right subtree and delete that node (which is now empty).



Expression Tree



- An expression tree is a Binary Tree which stores/represents the mathematical (arithmetic) expressions.
- The leaves of an expression tree are operands, such as constants or variable names and all the internal nodes are the operators.
- Formally we can define an expression Tree as a special kind of binary tree in which:
 - Each leaf is an operand. Examples: a, b, c, 6, 100
 - The root and internal nodes are operators. Examples: +, -, *, /, ^
 - Subtrees are subexpressions with the root being an operator.

EXAMPLE

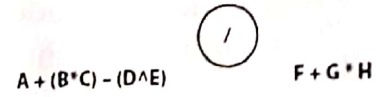
- Represent an Expression Tree

$$A + (B * C) - (D \wedge E) / F + G * H$$

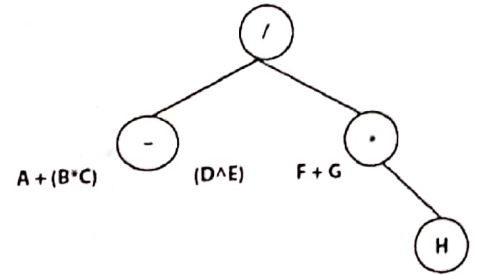
- choose an operator in such a way that the terms in parenthesis will be in a side

- Choose a operator having higher precedence. به اعلى
precedence. اولوية

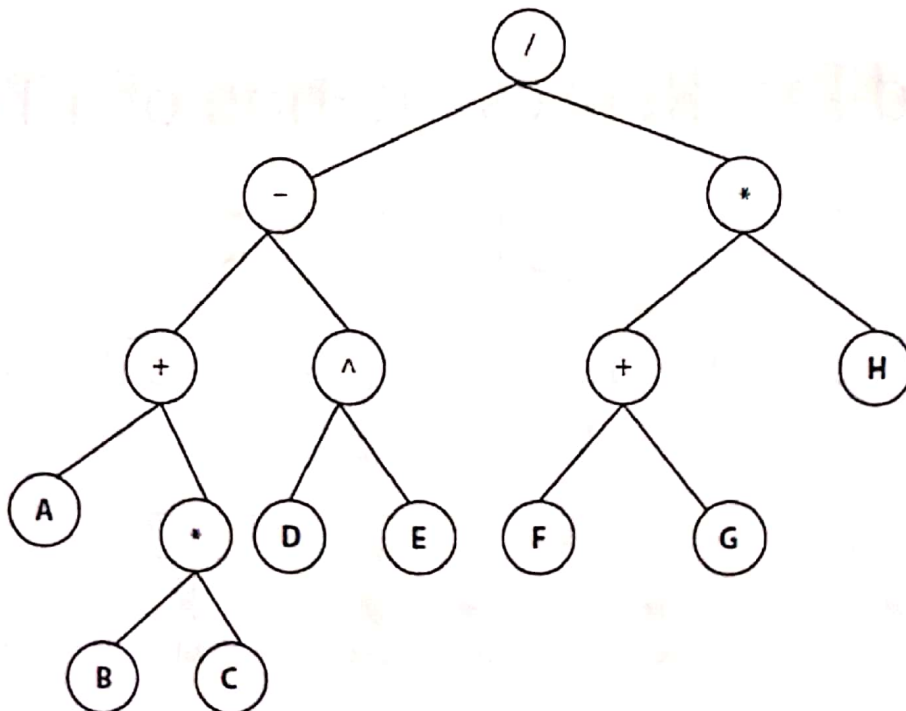
STEP1:



STEP2:



Step3:



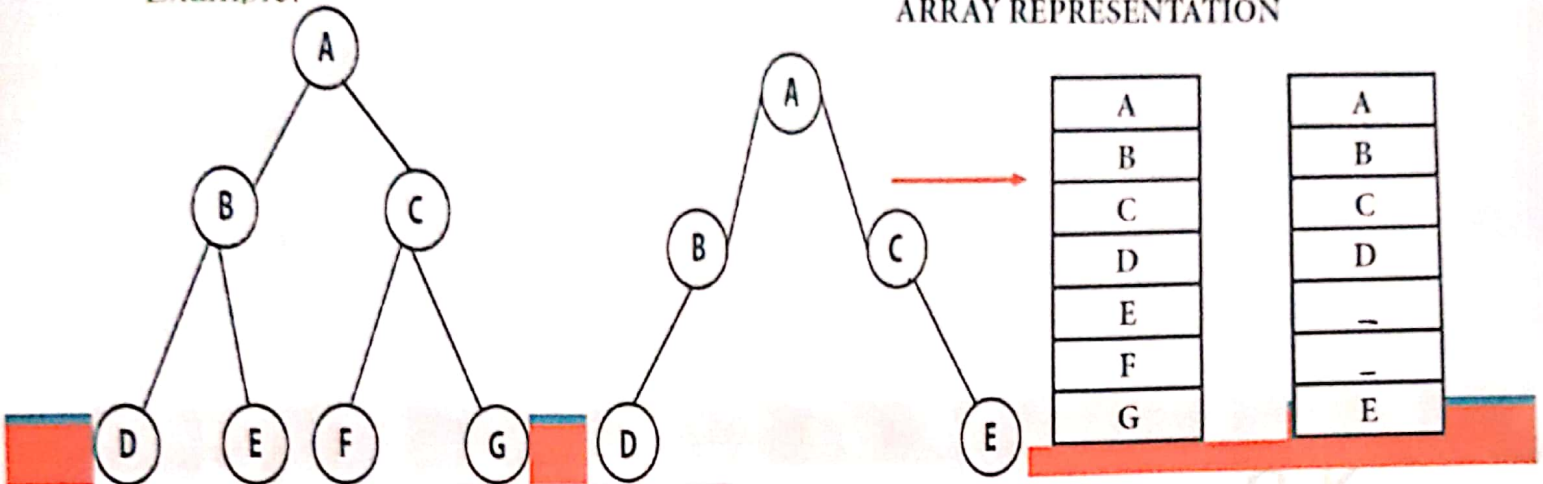


Array Representation of a Tree

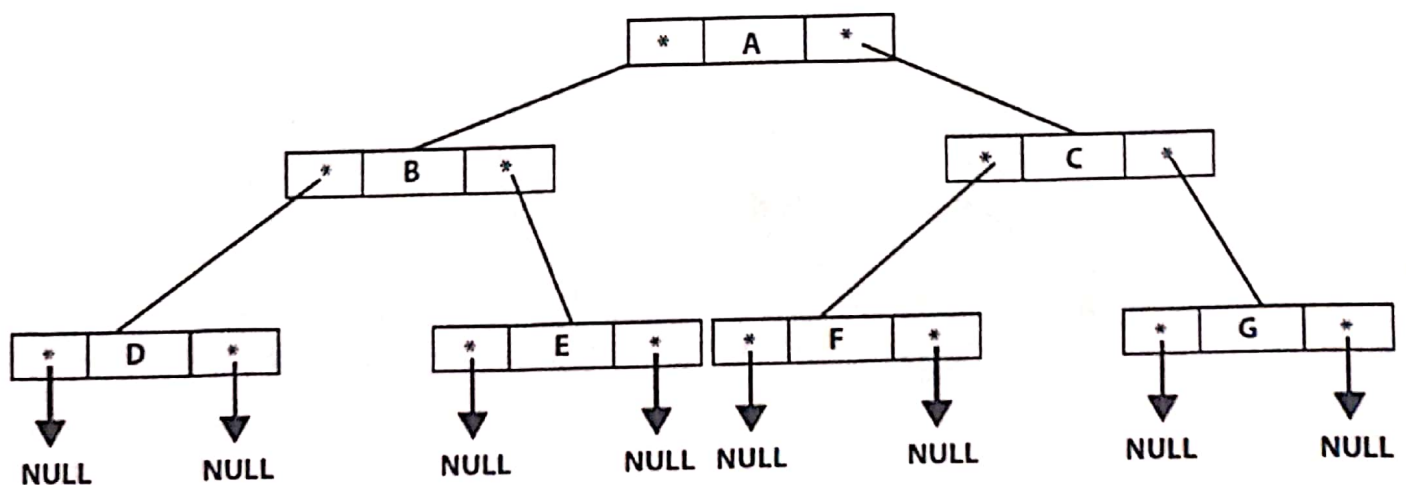
- The ROOT node is always kept as the FIRST element of the array i.e/ in the 0-Index the root node will be store. Then, in the successive memory locations the left child and right child are stored.

• Example:

ARRAY REPRESENTATION



Linked List Representation of a Tree



Binary Tree Traversal

Traversal:

The process of visiting all the elements in the data structure only once

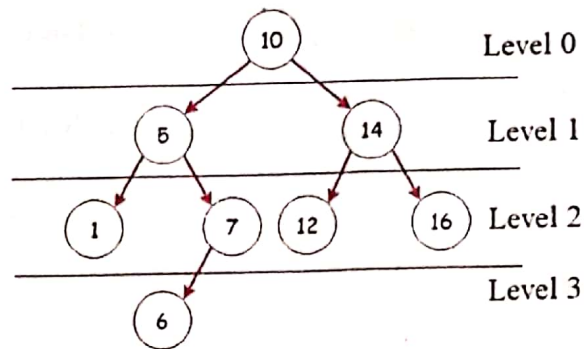
Traversal Techniques:

- Breadth First (Level-Order)

- Depth First

- 1) Preorder
- 2) Inorder
- 3) Postorder

Binary Tree Traversal

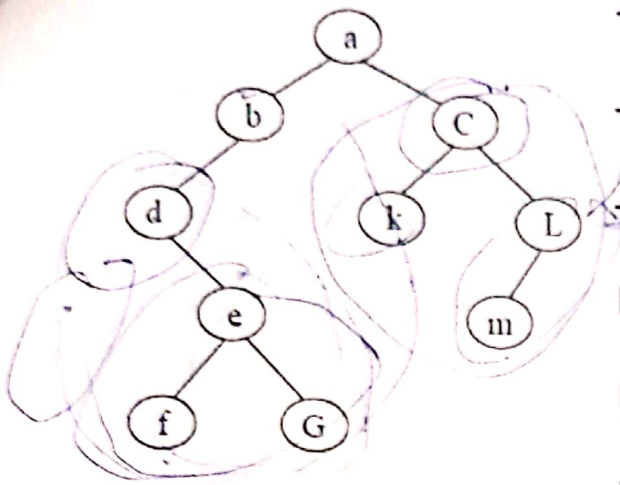


Flavors of (Depth First) Traversal

- In a *preorder traversal*, a node is visited before its descendants [root][left][right]
- In a *postorder traversal*, a node is visited after its descendants [left][right][root]
- In an *inorder traversal* a node is visited after its left subtree and before its right subtree [left][root][right]



Binary Tree Traversal -Example



-Preorder : [root][left][right]

-Inorder : [left][root][right]

-Postorder : [left][right][root]

Preorder : { a , b , d , e , f , G , C , k , L , m }

Inorder : { d , f , e , G , b , a , k , C , m , L }

Postorder : { f , G , e , d , b , k , m , L , C , a }

preorder : { a , b , d , e , f , G , C , k , L , m }



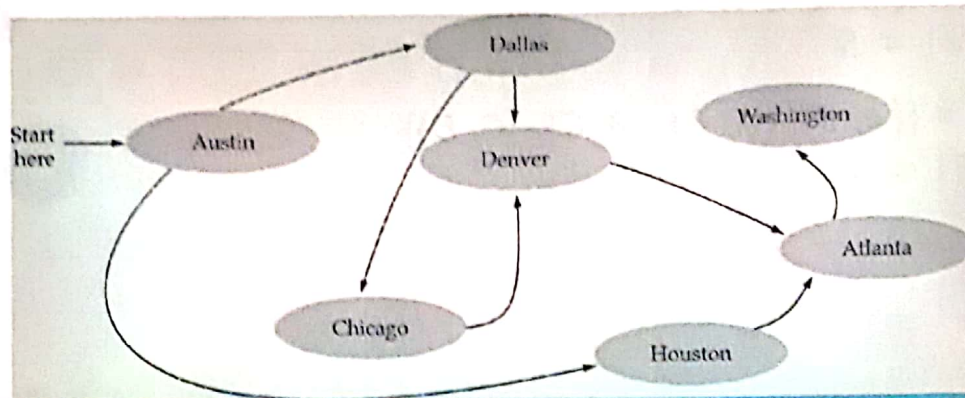
Algorithms and data structures-2

LECTURE-3+4 -GRAPH DATA STRUCTURE

ENG.TALAL SULTAN

What is a graph?

A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other. The set of edges describes relationships among the vertices.



Formal definition of graphs

A graph G is defined as follows:

$$G=(V,E)$$

$V(G)$: a finite, nonempty set of vertices

$E(G)$: a set of edges (pairs of vertices)



Directed vs. undirected graphs



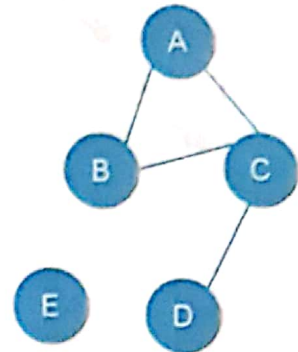
When the edges in a graph have no direction, the graph is called **undirected**

$$V = \{A, B, C, D, E\}$$

$$|V| = 5$$

$$E = \{\{A, B\}, \{A, C\}, \{B, C\}, \{C, D\}\}$$

$$|E| = 4$$



Directed vs. undirected graphs (cont.)



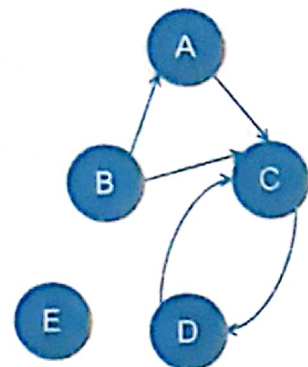
When the edges in a graph have a direction, the graph is called **directed** (or *digraph*)

$$V = \{A, B, C, D, E\}$$

$$E = \{(A, C), (B, A), (B, C), (C, D), (D, C)\}$$

$$|V| = 5$$

$$|E| = 5$$

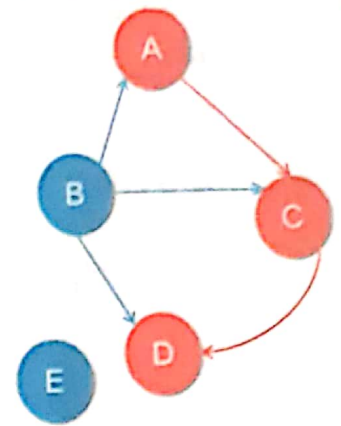


Graph terminologies : Paths

A **path** is a sequence $v_0, v_1, v_2, \dots, v_p$ of vertices such that for $0 \leq i < p$,

- Directed: $(v_i, v_{i+1}) \in E$
- Undirected: $\{v_i, v_{i+1}\} \in E$

The **length** of a path is its number of edges



Path: A,C,D

Convert undirected $\leftrightarrow$ directed?

Convert **undirected to directed** and maintain **paths**:

- Nodes unchanged
- Replace each edge $\{u,v\}$ with two edges $\{(u,v), (v,u)\}$

Convert **directed to undirected** and maintain paths:

Can't: why?

Because any path from A to B in undirected graph would be reversible to a path from B to A, which does not exist in original graph



More Graph terminology

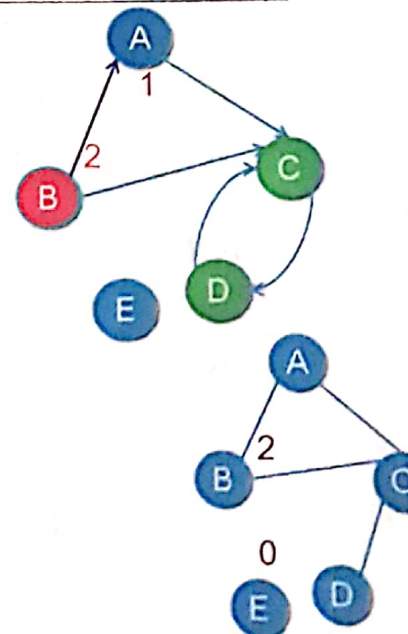
Adjacent nodes: A node 'V' is said to be **adjacent node** of node 'U' if and only if there exists an edge between 'U' and 'V'.



V is adjacent of U

More Graph terminology

- Vertices u and v are called
 - ▣ the **source** and **sink** of the directed edge (u, v) , respectively
 - ▣ the **endpoints** of (u, v) or $\{u, v\}$
- The **outdegree** of a vertex u in a directed graph is the number of edges for which u is the source
- The **indegree** of a vertex v in a directed graph is the number of edges for which v is the sink
- The **degree** of a vertex u in an undirected graph is the number of edges of which u is an endpoint



More Graph terminology

Pendent Vertex: A vertex with degree one.



Isolated Vertex: A vertex with degree zero.

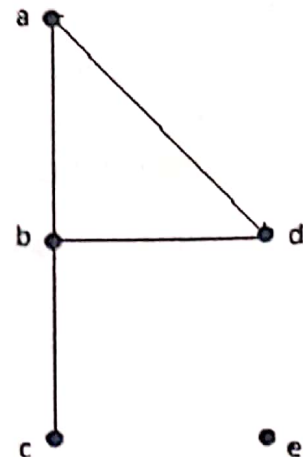


Loop: An edge of a graph that starts from a vertex and ends at the same vertex is called a loop or a self-loop.

10

Examples

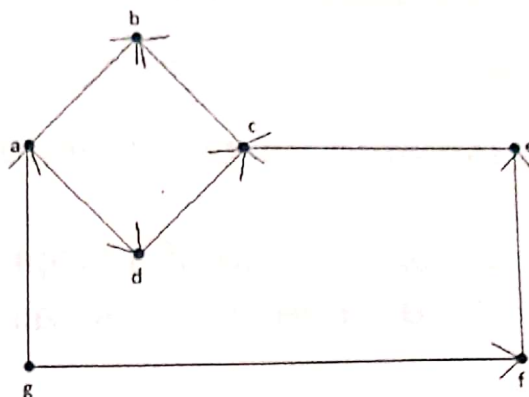
$\deg(a) = 2$
 $\deg(b) = 3$
 $\deg(c) = 1$
 $\deg(d) = 2$
 $\deg(e) = 0$



Examples



Vertex	Indegree	Outdegree
a	1	2
b	2	0
c	2	1
d	1	1
e	1	1
f	1	1
g	0	2

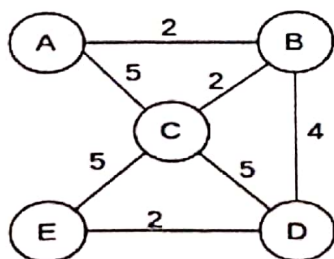


More Graph terminology

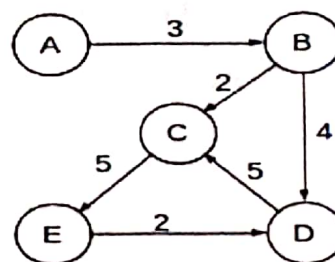


Weighted Graphs: A graph in which edges have weights or costs associated with them.

- Example: A road network graph where the weights can represent the distance between two cities.



Undirected

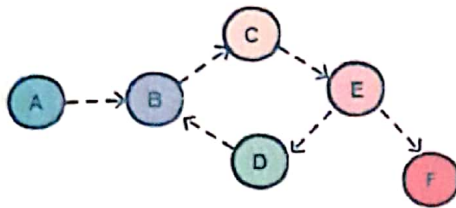


Directed

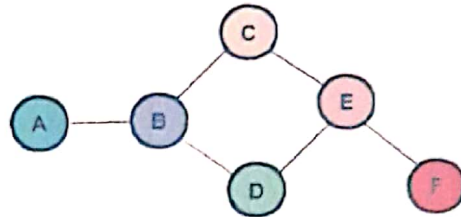
More Graph terminology

Unweighted Graphs: A graph in which edges have no weights or costs associated with them.

- Example: A social network graph where the edges represent friendships.



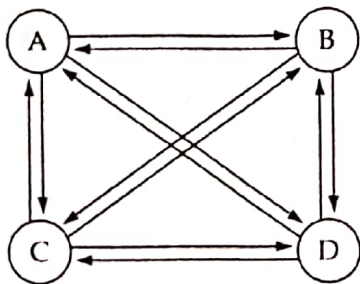
Directed Graph



Undirected Graph

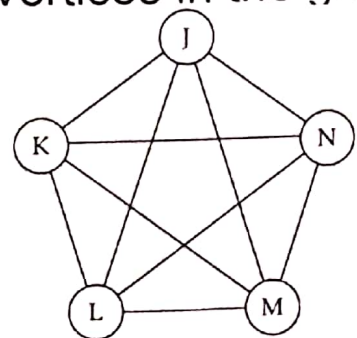
More Graph terminology

Complete Graph: graph with n vertices is called a complete graph if the degree of each vertex is $n-1$, that is, one vertex is attached with $n-1$ edges or the rest of the vertices in the graph.



(a) Complete directed graph.

$$N * (N-1)$$



(b) Complete undirected graph.

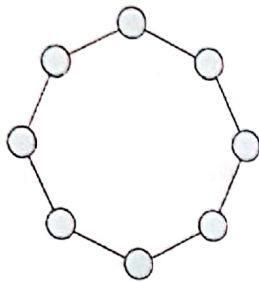
$$N * (N-1) / 2$$



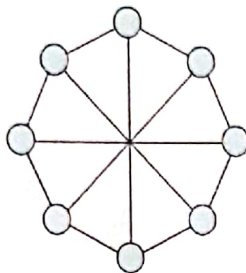
More Graph terminology



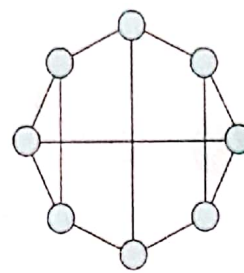
Regular Graph: graph is said to be regular if all vertices of graph G are of equal degree.



2-regular



3-regular



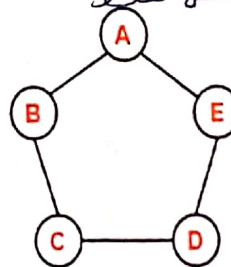
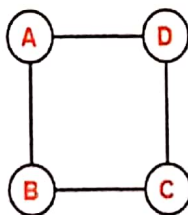
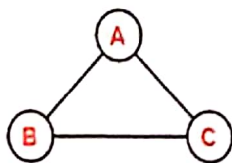
3-regular

More Graph terminology



Cycle Graph: If the degree of each vertex in the graph is two, then it is called a Cycle Graph.

regular cycle graph or
دورة



Examples of Cycle Graph

More Graph terminology

Dense Graph: A dense graph is a graph in which there is a large number of edges. Typically in a dense graph, the number of edges is close to the maximum number of edges.

Sparse Graph: A sparse graph is a graph in which there is a small number of edges. In this case the number of edges is considerably less than the maximum number of edges.

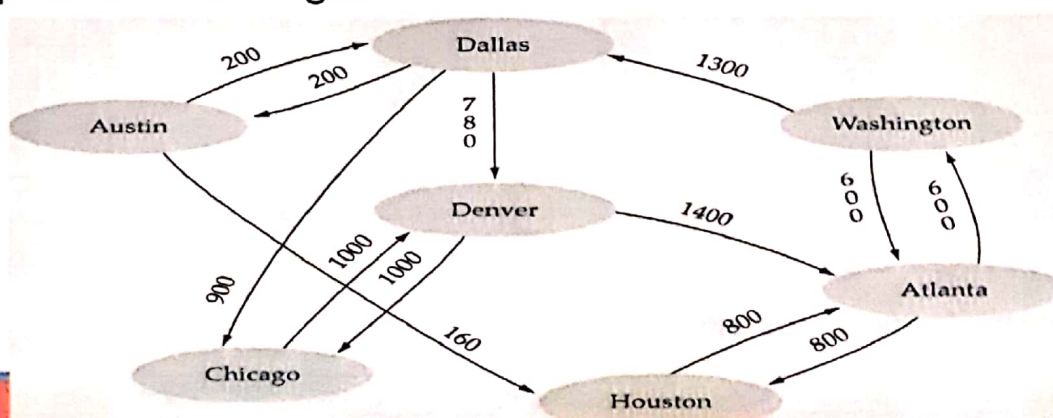
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]	0	1	1	0	0	0	0
[1]	1	0	0	1	0	0	0
[2]	1	0	0	0	0	1	0
[3]	0	1	0	0	0	1	0
[4]	1	0	0	1	0	0	1
[5]	0	0	1	1	0	0	0
[6]	0	0	0	0	1	0	0

Atlanta [0]
Washington [1]
Houston [2]
Dallas [3]
Denver [4]
Austin [5]
Chicago [6]

Graph implementation

Array-based implementation

- A 1D array is used to represent the vertices
- A 2D array (adjacency matrix) is used to represent the edges





Array-based implementation example



graph

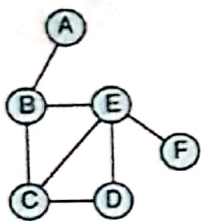
.numVertices 7

.vertices

.edges

[0]	"Atlanta"	"
[1]	"Austin"	"
[2]	"Chicago"	"
[3]	"Dallas"	"
[4]	"Denver"	"
[5]	"Houston"	"
[6]	"Washington"	"

[0]	0	0	0	0	0	800	600
[1]	0	0	0	200	0	160	0
[2]	0	0	0	0	1000	0	0
[3]	0	200	900	0	780	0	0
[4]	1400	0	1000	0	0	0	0
[5]	800	0	0	0	0	0	0
[6]	600	0	0	1300	0	0	0



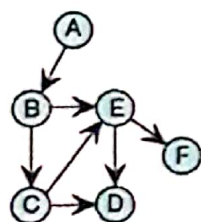
A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Adjacency matrix

(a) Adjacency matrix for non-directed graph



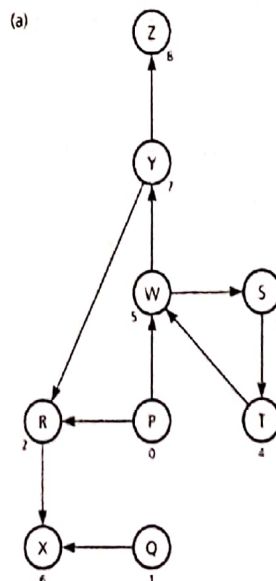
A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

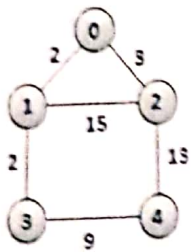
Adjacency matrix

(a) Adjacency matrix for directed graph



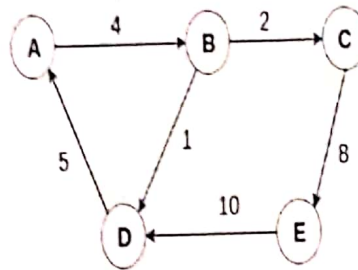
	0	1	2	3	4	5	6	7	8
P	0	0	1	0	0	1	0	0	0
Q	0	0	0	0	0	0	1	0	0
R	0	0	0	0	0	0	1	0	0
S	0	0	0	0	1	0	0	0	0
T	0	0	0	0	0	1	0	0	0
W	0	0	0	1	0	0	0	1	0
X	0	0	0	0	0	0	0	0	0
Y	0	0	1	0	0	0	0	0	1
Z	0	0	0	0	0	0	0	0	0

Adjacency matrix Examples



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix Representation of Weighted Graph



Weighted Directed Graph

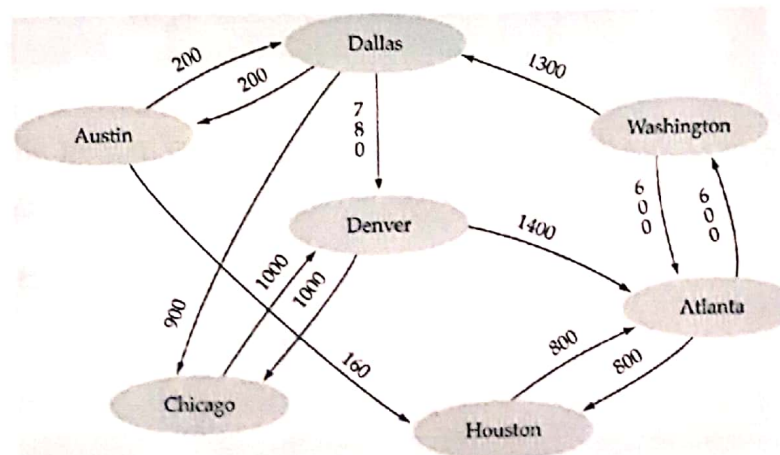
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

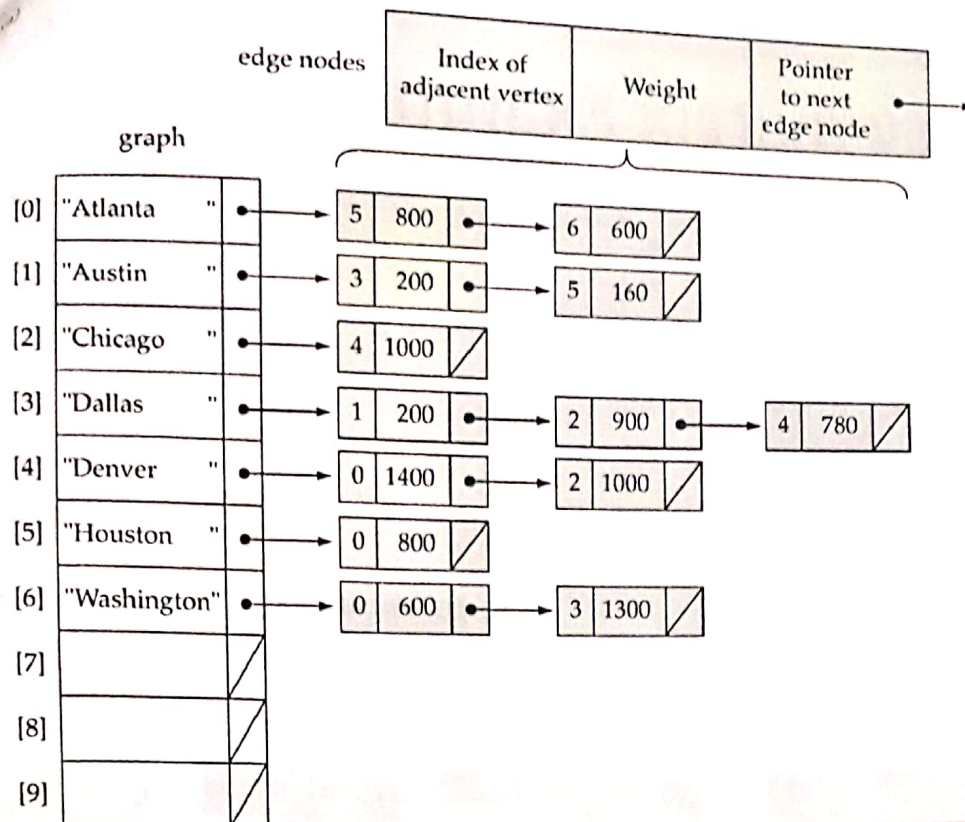
Adjacency Matrix

Graph implementation (cont.)

Linked-list implementation

- A 1D array is used to represent the vertices
- A list is used for each vertex v which contains the vertices which are adjacent from v (adjacency list)





Adjacency list vs. Adjacency matrix

List	Property	Matrix
$O(V + E)$	Space	$O(V ^2)$
$O(V + E)$	Time to visit all edges	$O(V ^2)$
$O(V + \text{od}(v_i))$	Time to find edge (v_1, v_2)	$O(1)$

Sparse graphs

Better for

Dense graphs



Max edges =

Sparse: $|E| \ll |V|^2$

Dense: $|E| \approx |V|^2$

Graph traversal

Problems:

- Finding all the nodes which are reachable
- Finding the best path through a graph And so on. The main goal of this type of graph traversal is to find all the nodes that are reachable from a given set of root nodes.

Methods:

- Depth-First-Search (DFS)
- Breadth-First-Search (BFS)

What is BFS and DFS?

Breadth First Search (BFS) traversal is mainly used to find the shortest path in an unweighted graph. In order to perform the BFS, we need to implement the Queue Data Structure, which follows the principle of **FIFO (First In First Out)**.

Depth First Search (DFS) traversal mainly traverse the graph in a depthward motion. This traversal is used for cycle detection, strongly connected components and so on. We can implement DFS by using the Stack Data structure, which follows **LIFO (Last In First Out)** Principle.



Steps to implement Depth First Traversal(DFS)



Step 1 – Visit adjacent unvisited vertex. Mark it as visited. Print it and Push it in a stack.

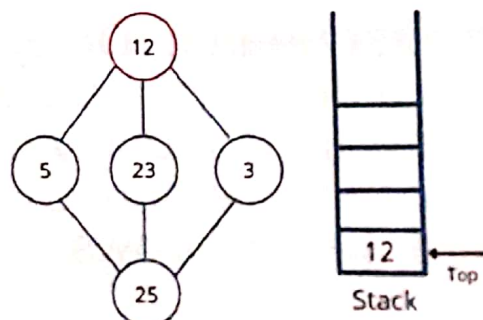
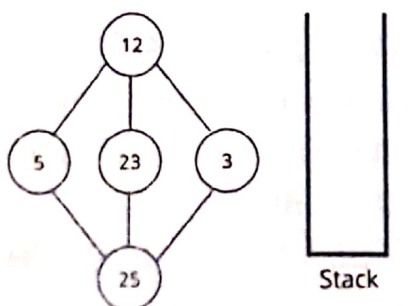
Step 2 – If no adjacent vertex is found, pop up a vertex from the stack.

Step 3 – Repeat Step 1 and Step 2 until the stack is empty.

Steps to implement Depth First Traversal



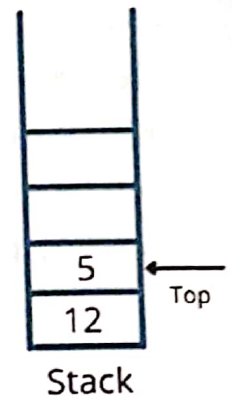
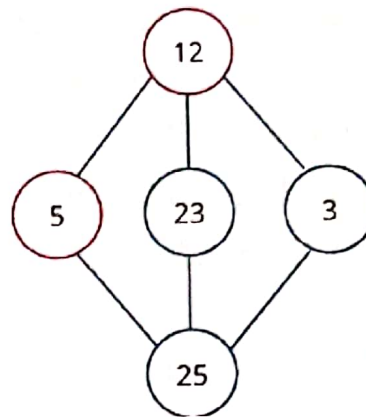
1. Initialize the stack
2. Mark **12** as visited and push into stack



Steps to implement Depth First Traversal

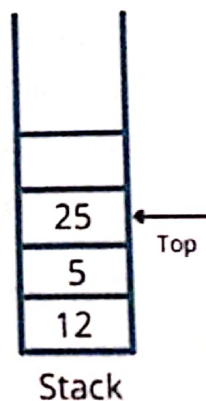
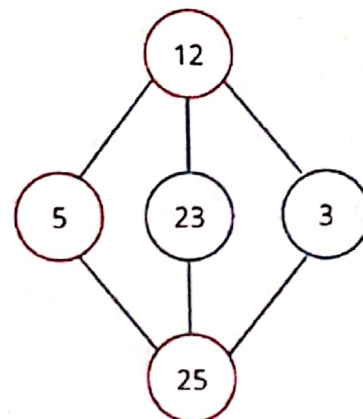
Now Explore any unvisited adjacent node from **12**. In our example We have three nodes. We can pick any of them. Here we are going to pick **5**.

Mark **5** as visited and put it onto the stack. Explore any unvisited adjacent node from **5**. Both **12** and **25** are adjacent to **5** but we are concerned for unvisited nodes only.



Steps to implement Depth First Traversal

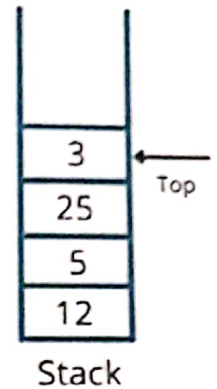
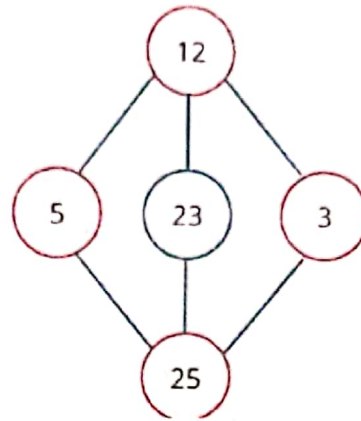
Visit **25** and mark it as visited and put onto the stack. Here, we have **23** and **3** nodes, which are adjacent to **25** and both are unvisited.



Steps to implement Depth First Traversal



We choose **3**, mark it as visited and put onto the stack. Here **3** does not have any unvisited adjacent node. So, we pop **3** from the stack.

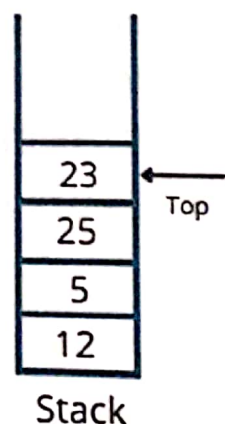
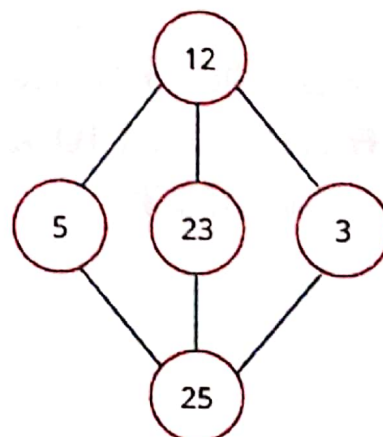


Steps to implement Depth First Traversal

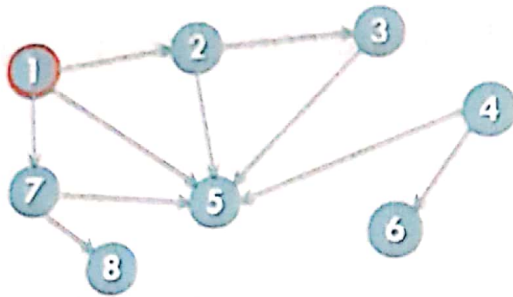


We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find 25 to be on the top of the stack.

Only unvisited adjacent node is from 25 is 23 now. So we visit 23, mark it as visited and put it onto the stack



DFS Example



dfs(1) visits the nodes in this order: 1, 2, 3, 5, 7, 8

Steps to implement BFS traversal

Step 1 – First define a Queue

Step 2 – Select any vertex which is a starting point from where traversal will start.

Step 3 – Visit starting vertex and insert it into the Queue.

Step 4 – Visit all the non-visited adjacent vertices which is connected to it and insert all non-visited vertices into the Queue.

Step 5 – When there is no new vertex to be visited from the element which is top in a queue, Remove the top element which is the vertex from the queue.

Step 6 – Repeat steps 4 and 6 until the queue becomes empty.

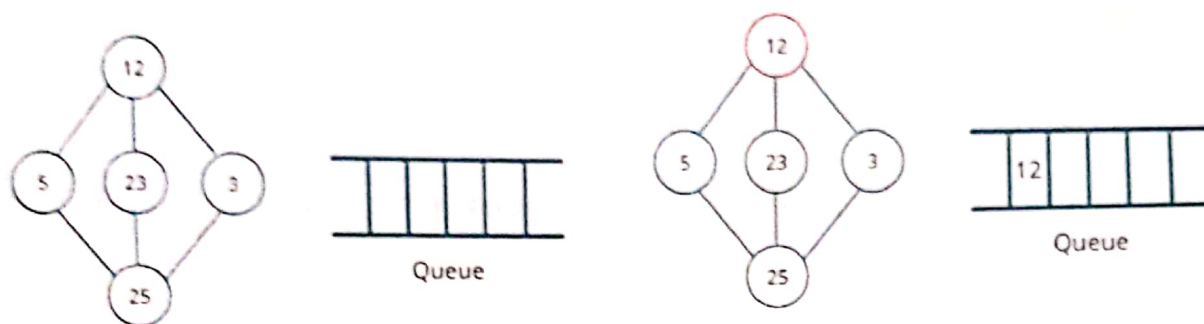




Steps to implement BFS traversal

Initialize the queue

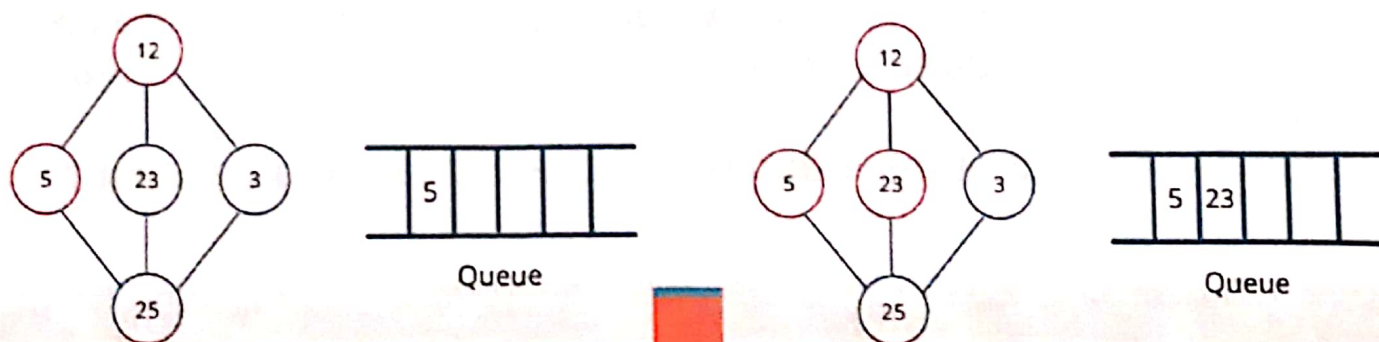
We start visiting from vertex **12** that can be considered as starting node, and mark it as visited.



Steps to implement BFS traversal

Now we will see an unvisited adjacent node from **12**. In this example, we have three nodes and we can visit anyone. here we are traversing from left to right. We choose **5** and mark it as visited and enqueue it.

Next, visit the unvisited adjacent node from **12** to **23**. We mark it as visited and enqueue it.

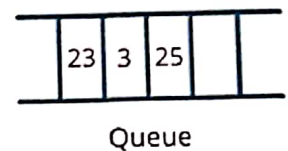
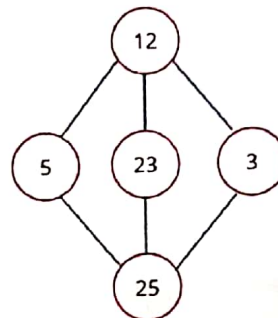
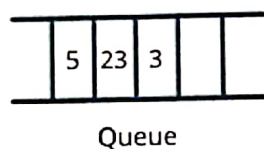
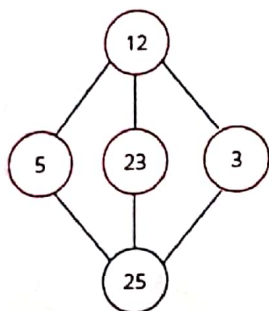


Steps to implement BFS traversal

Next, the unvisited adjacent node from **12** is **3**. We mark it as visited and enqueue it.

Now, all the connected adjacent node from **12** is traversed and no unvisited adjacent nodes left. So, we dequeue and find all connect vertex from **5**.

From **5** we have **25** as unvisited adjacent node. We mark it as visited and enqueue it.



Steps to implement BFS traversal

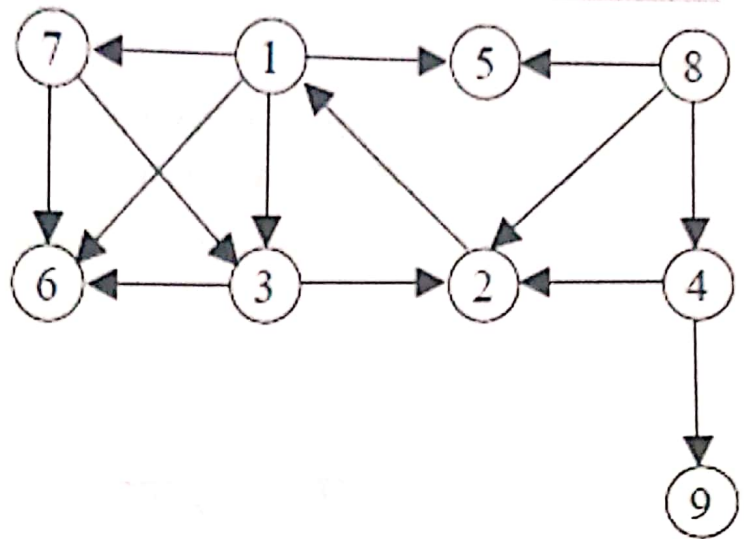
Now if all nodes visited, we will dequeue all nodes from queue.

So BFS Traversal output is : 12, 5, 23, 3, 25



DFS-BFS Example

Give the BFS and DFS Traversal output of this graph ,choose vertex (1) as start vertex.



BFS vs DFS

Parameters	BFS	DFS
Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
Technique	BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
Suitability for Decision-Trees	BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, and the then explore all paths through this decision. And if this decision leads to win situation, we stop.

BFS vs DFS



Parameters	BFS	DFS
Time Complexity	The Time complexity of BFS is $O(V + E)$ when Adjacency List is used and $O(V^2)$ when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also $O(V + E)$ when Adjacency List is used and $O(V^2)$ when Adjacency Matrix is used, where V stands for vertices and E stands for edges.
Visiting of Siblings/ Children	Here, siblings are visited before the children.	Here, children are visited before the siblings.
Removal of Traversed Nodes	Nodes that are traversed several times are deleted from the queue.	The visited nodes are added to the stack and then removed when there are no more nodes to visit.
Backtracking	In BFS there is no concept of backtracking.	DFS algorithm is a recursive algorithm that uses the idea of backtracking
Optimality	BFS is optimal for finding the shortest path.	DFS is not optimal for finding the shortest path.
When to use?	When the target is close to the source, BFS performs better.	When the target is far from the source, DFS is preferable.



264783



Some Graph-Processing Problems

Path - Is there a path between s and t ?

Shortest path – What is the shortest path between s and t ?

Cycle - Is there a cycle in the graph?

Euler tour – Is there a cycle that uses each edge exactly once?

Hamilton tour – Is there a cycle that uses each vertex exactly once?

Connectivity – Is there a way to connect all of the vertices?

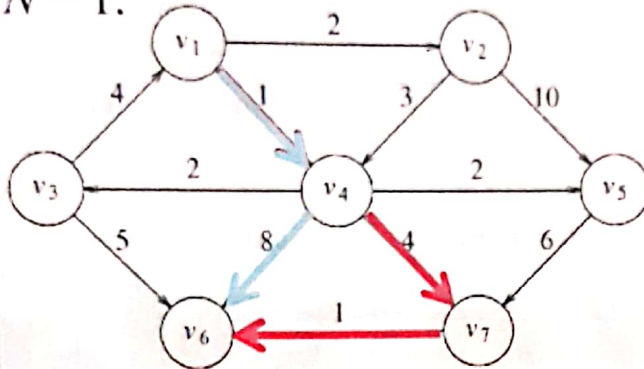
MST – What is the best way to connect all the vertices?

Shortest-Path Algorithms

The input is a weighted graph. Associated with each edge (v_i, v_j) is a cost c_{ij} to traverse the edge. The cost of a path $v_1 v_2 \dots v_N$ is $\sum_{i=1}^{N-1} c_{i, i+1}$

This is referred to as the **weighted path length**.

The **unweighted path length** is merely the number of edges on the path, namely, $N - 1$.

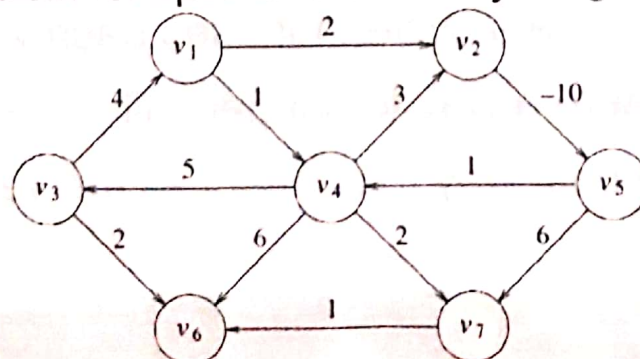


- the shortest weighted path from v_1 to v_6 has a cost of 8
- The shortest unweighted path between these vertices is 2.

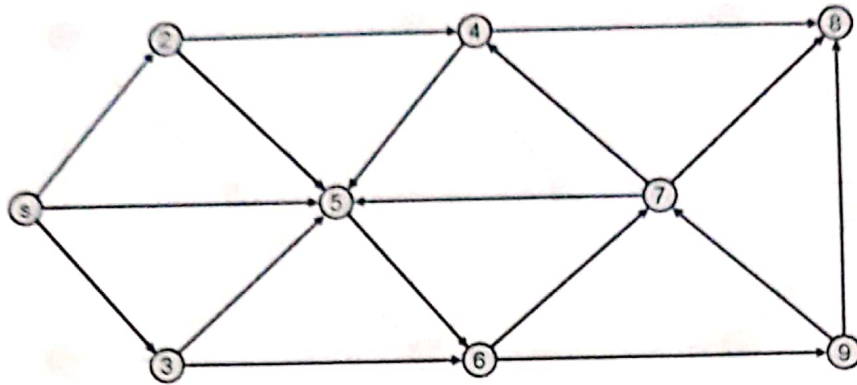
Shortest-Path Algorithms

Four versions of the problem of shortest-path:

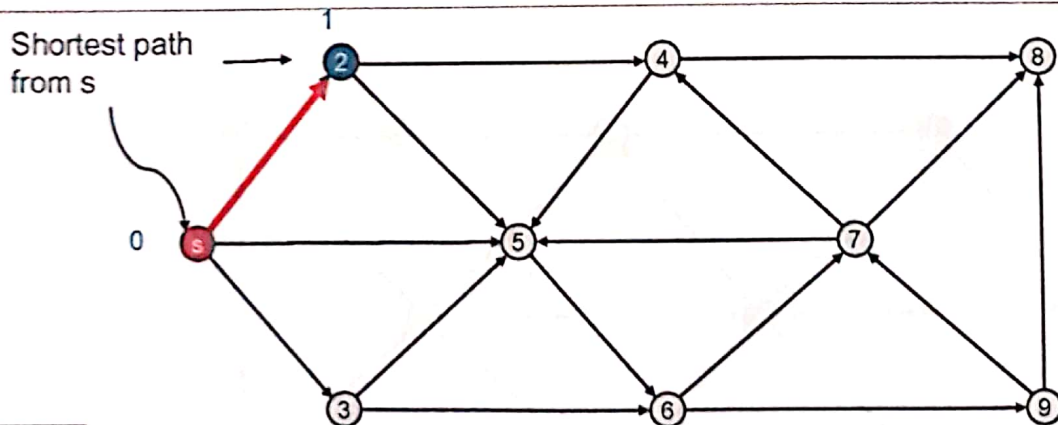
- Unweighted shortest-path problem;
- Weighted shortest-path with no negative edges problem;
- Weighted shortest-path with negative edges problem;
- Weighted problem for special case of acyclic graphs.



Unweighted shortest-path Breadth First Search



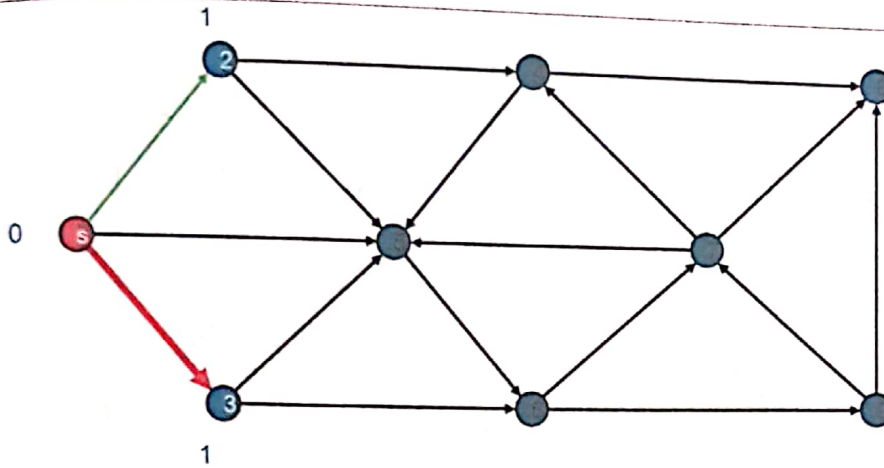
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: s

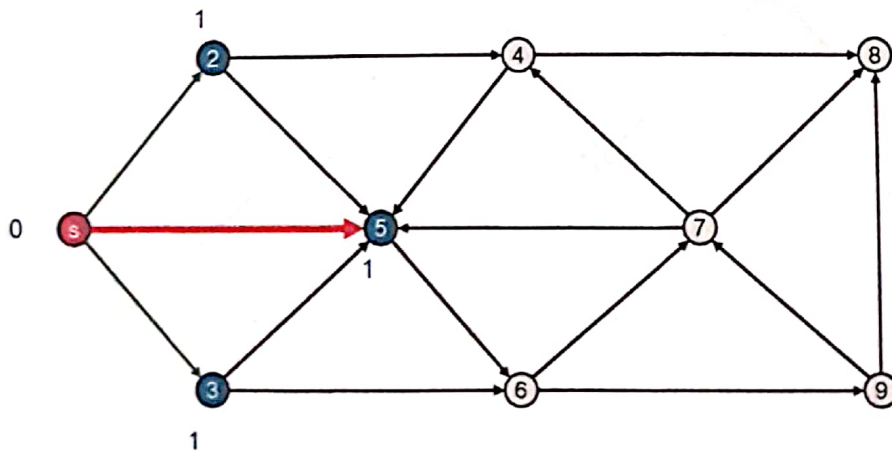
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: s 2

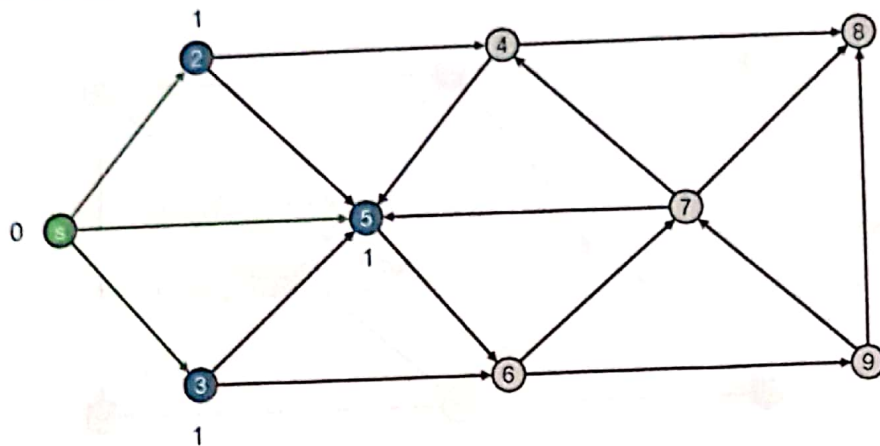
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: s 2 3

Breadth First Search

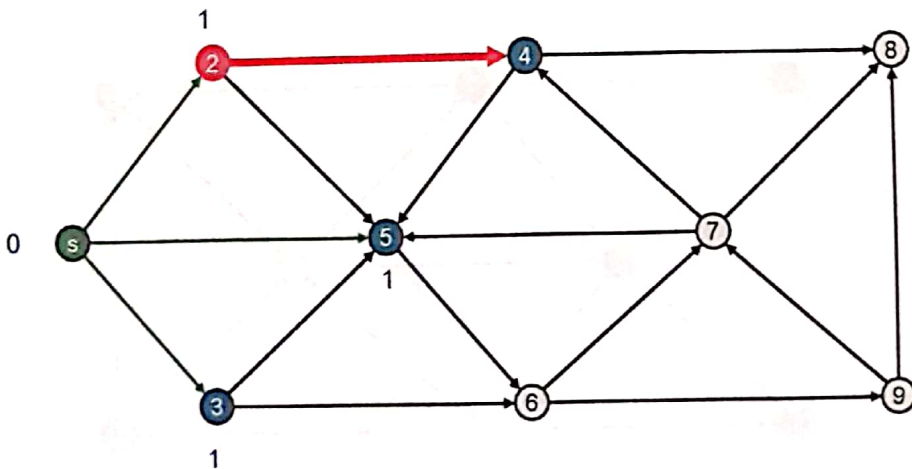


Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5



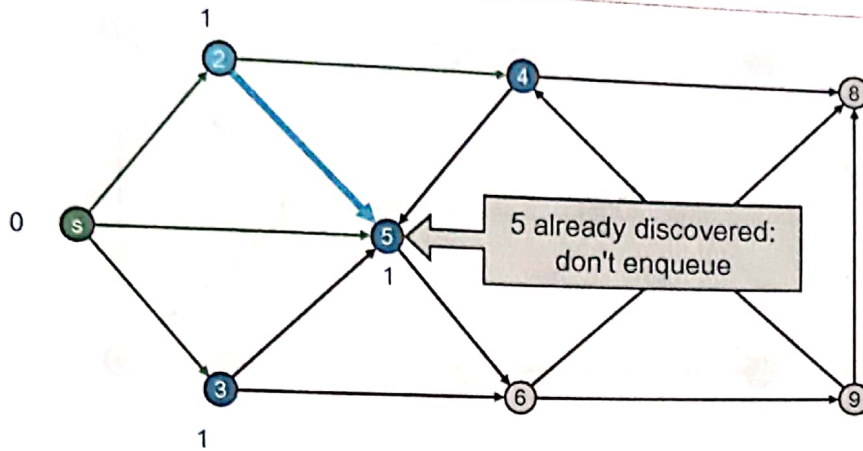
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5

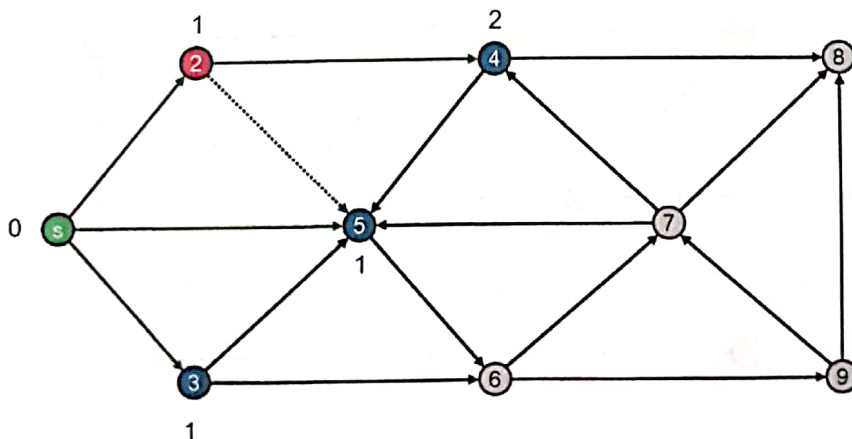
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5 4

Breadth First Search

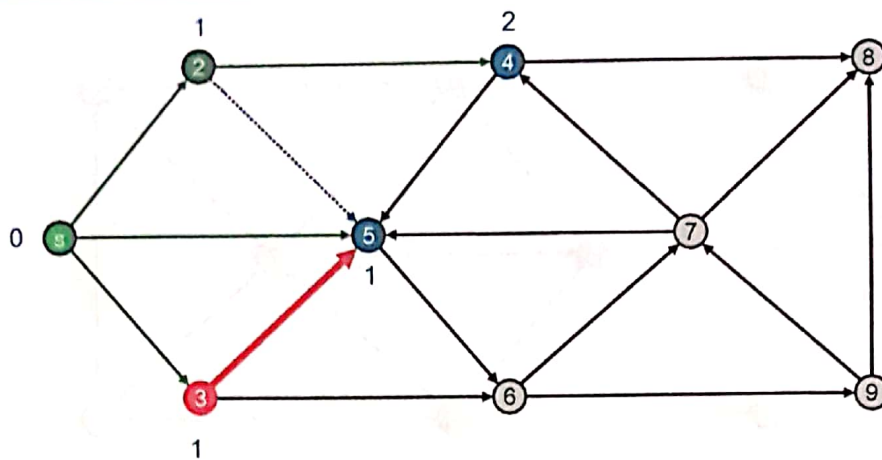


Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5 4



Breadth First Search

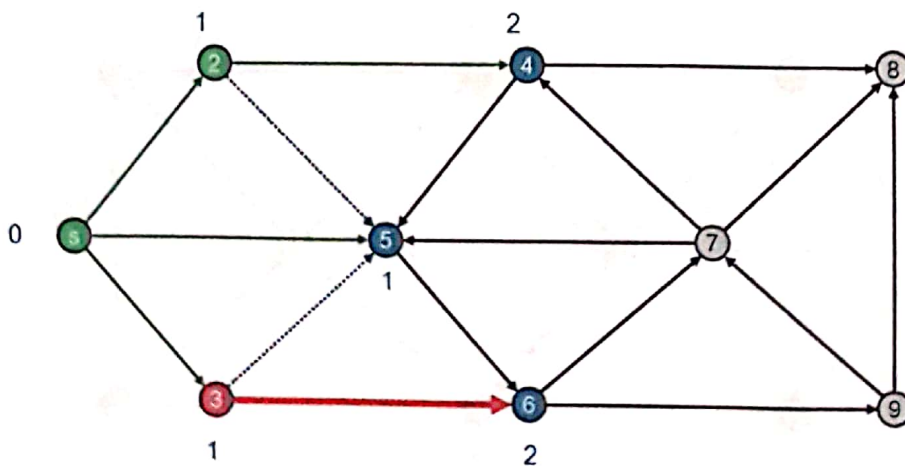


Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4



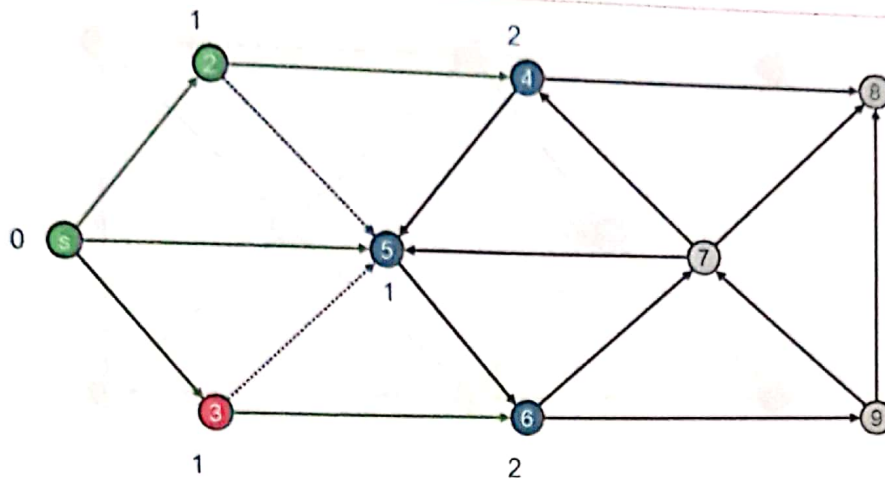
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4

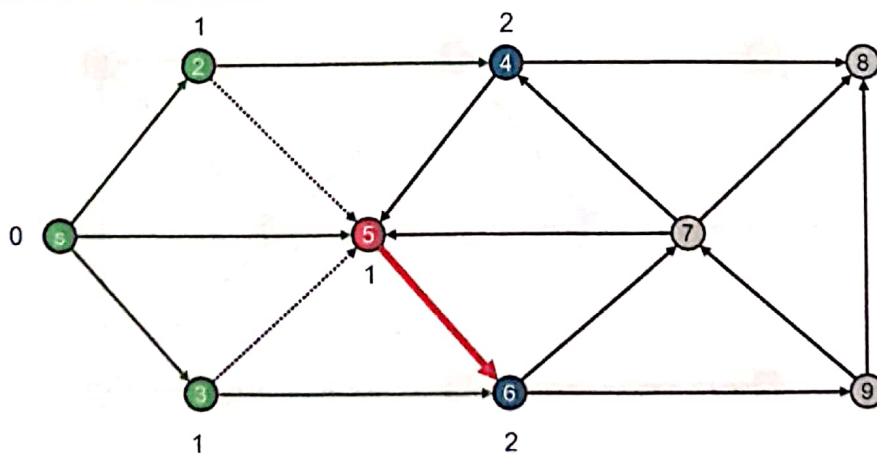
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4 6

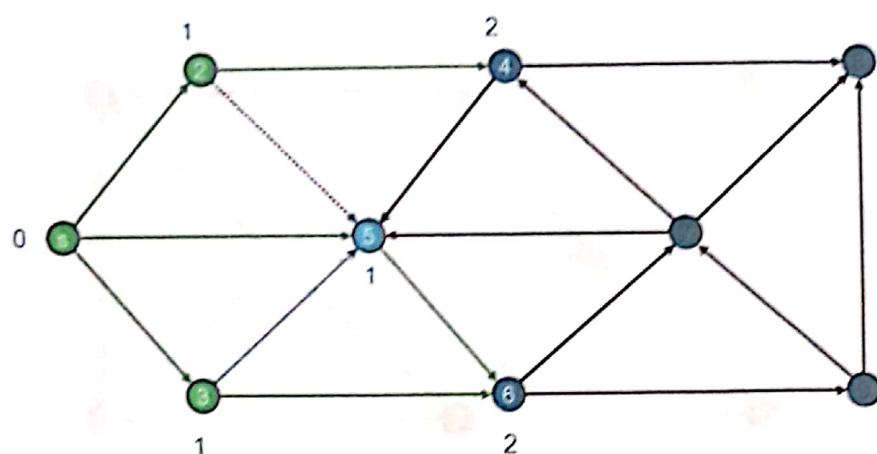
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 5 4 6

Breadth First Search



Undiscovered

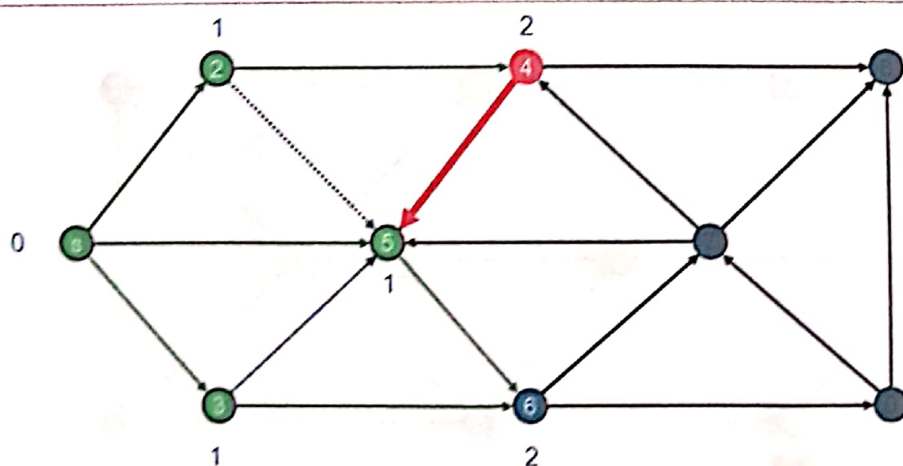
Discovered

Top of queue

Finished

Queue: 5 4 6

Breadth First Search



Undiscovered

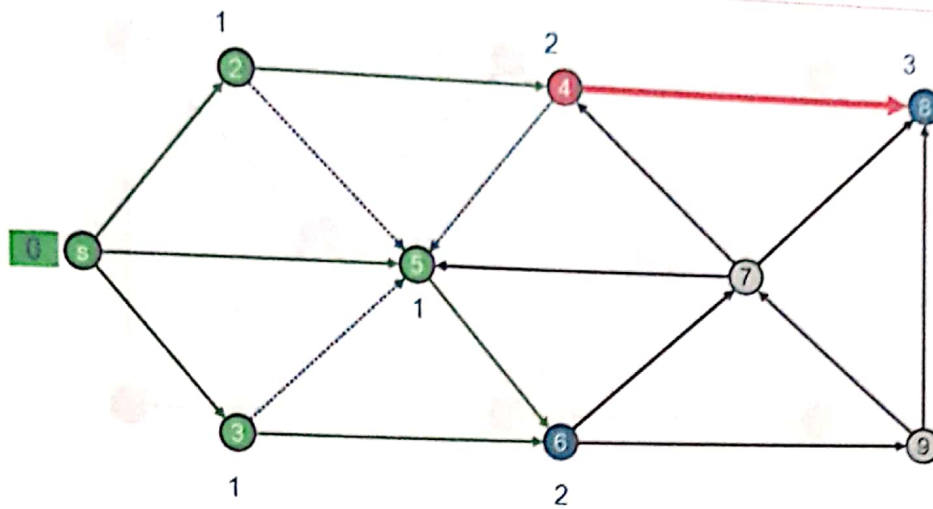
Discovered

Top of queue

Finished

Queue: 4 6

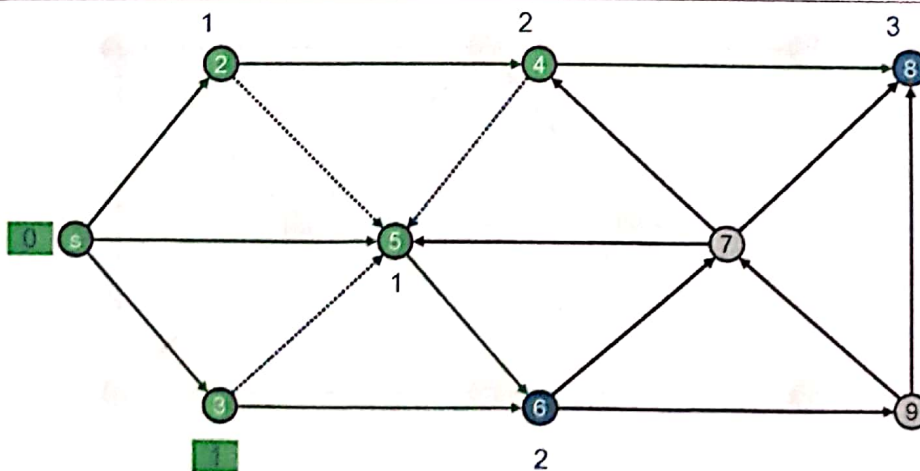
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6

Breadth First Search

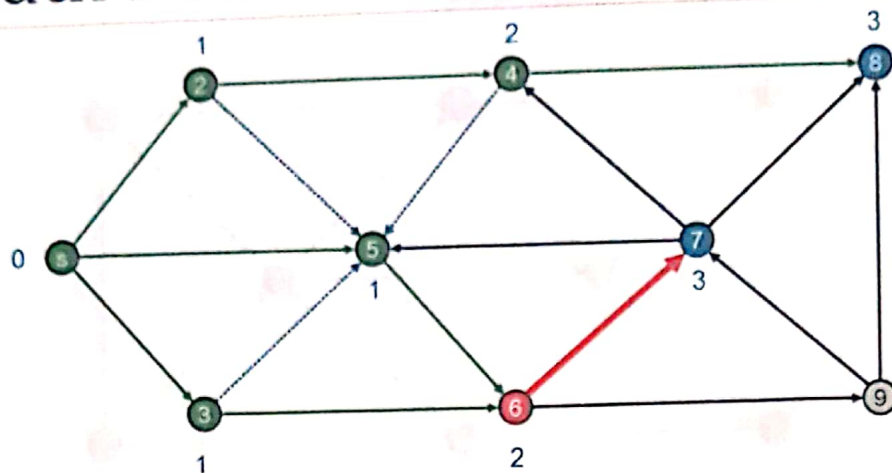


Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6 8



Breadth First Search

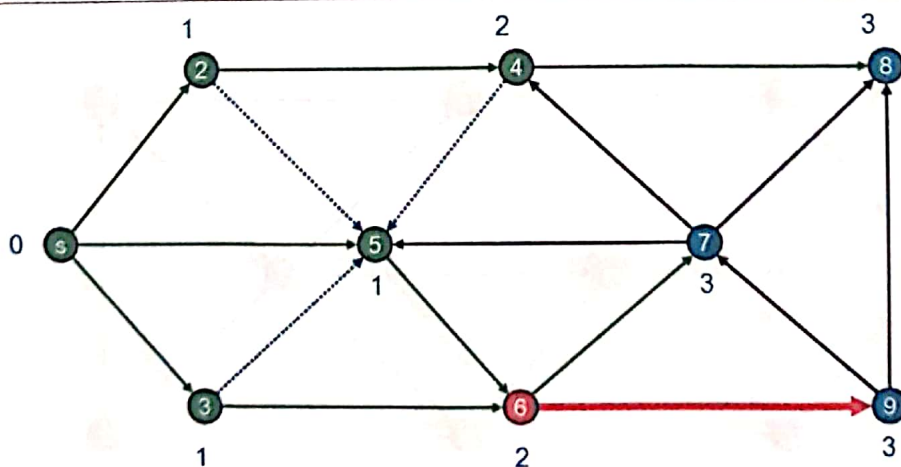


Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8



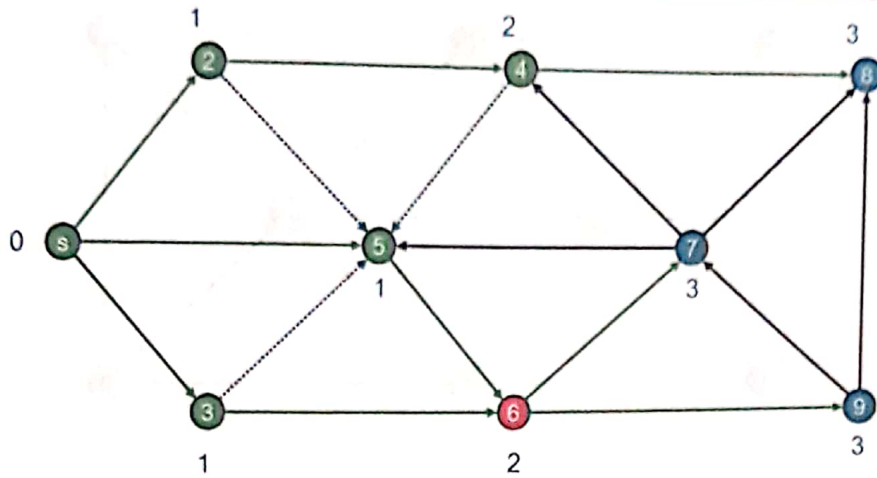
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7

Breadth First Search



Undiscovered

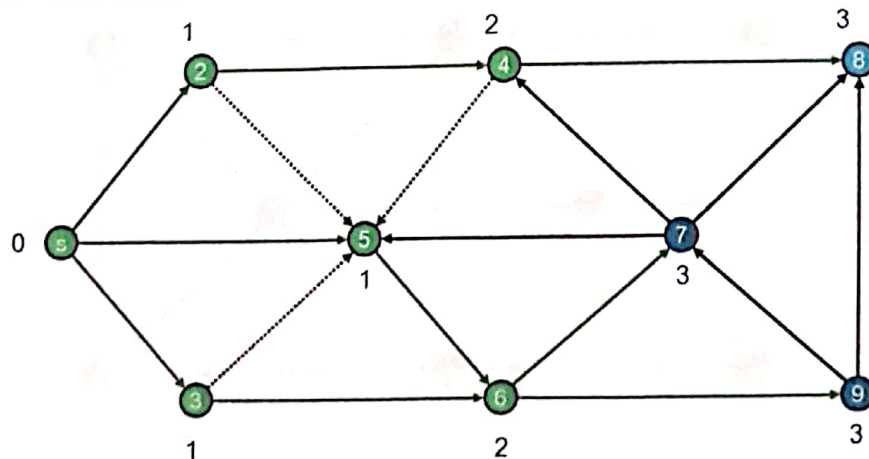
Discovered

Top of queue

Finished

Queue: 6 8 7 9

Breadth First Search



Undiscovered

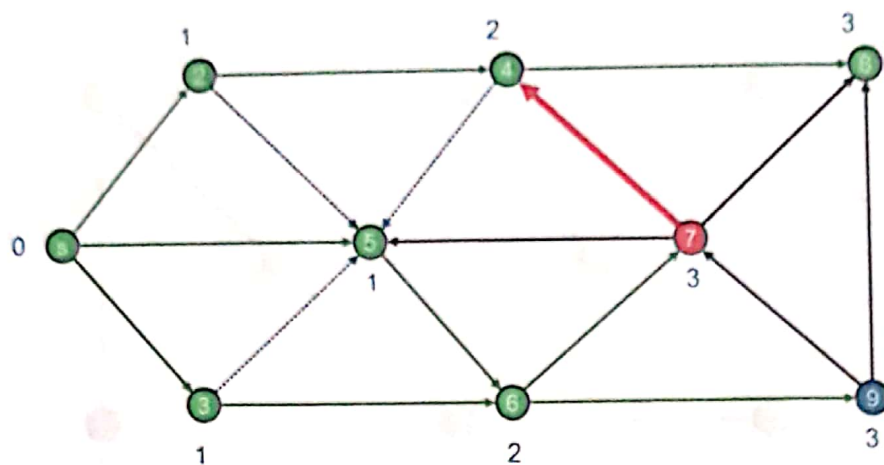
Discovered

Top of queue

Finished

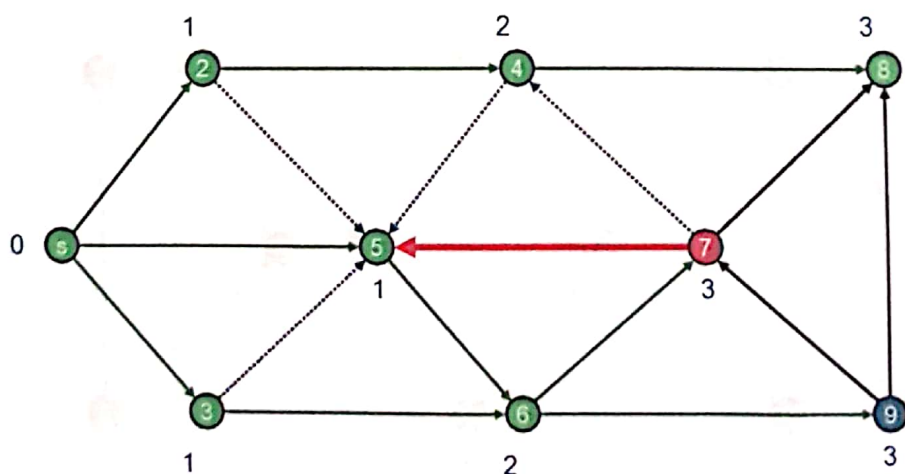
Queue: 8 7 9

Breadth First Search



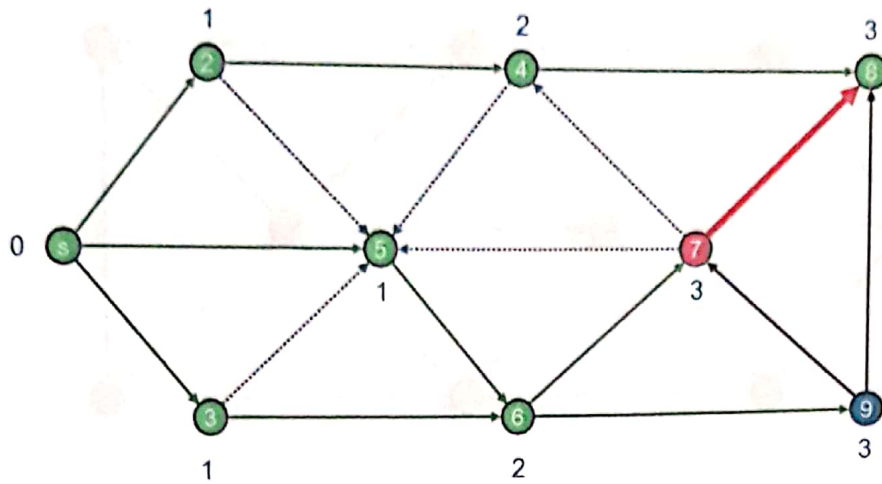
Queue: 7 9

Breadth First Search



Queue: 7 9

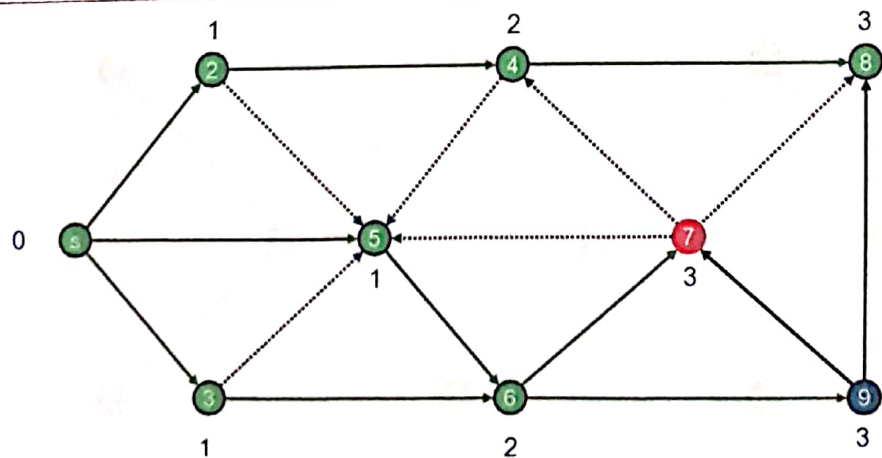
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

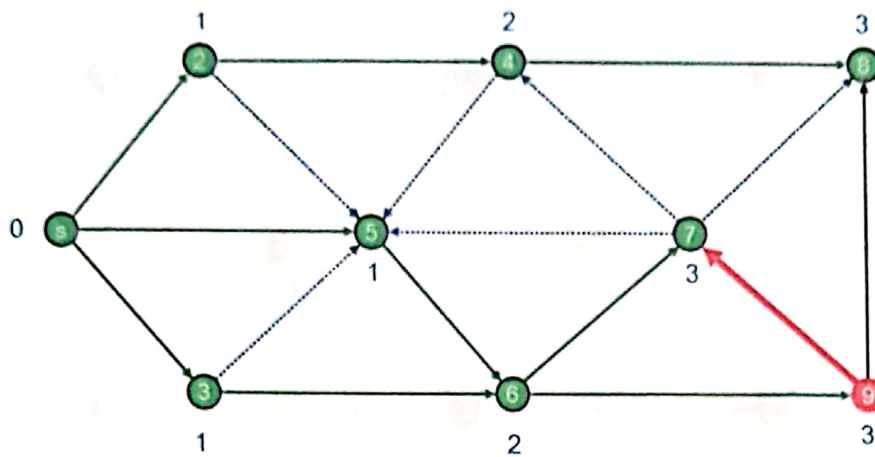
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

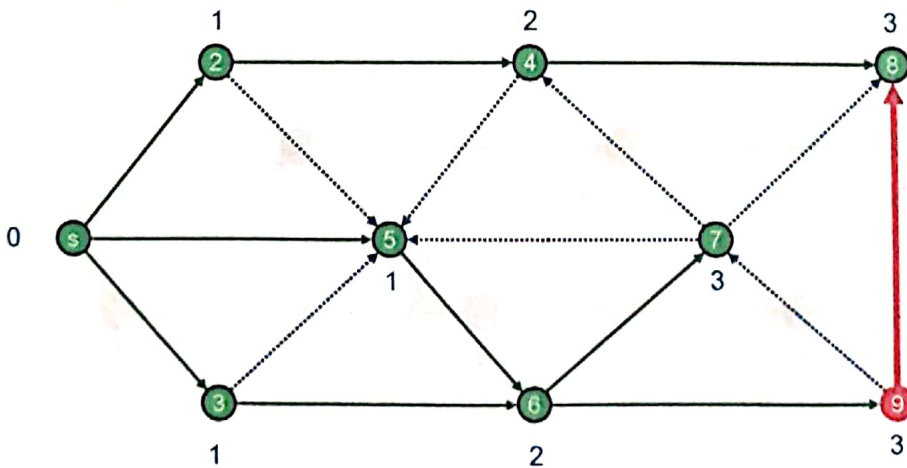
Breadth First Search



Queue: 9

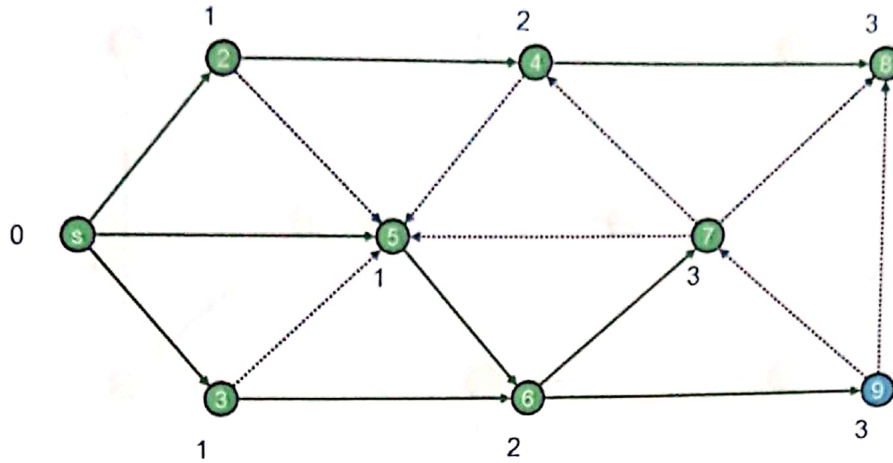


Breadth First Search



Queue: 9

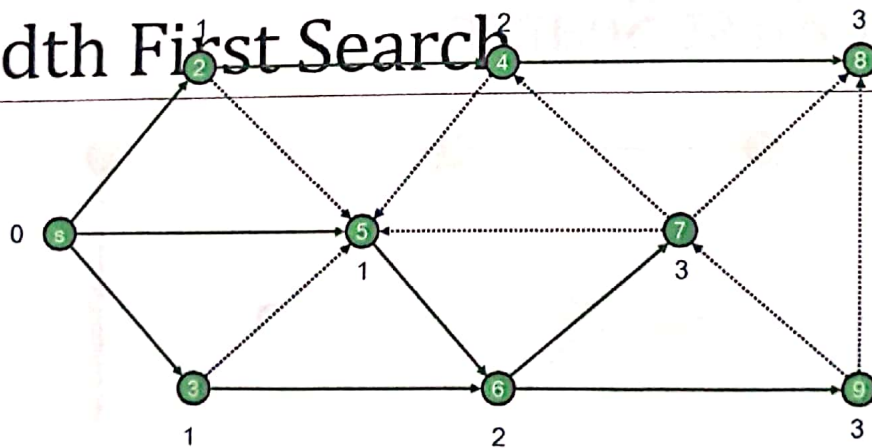
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

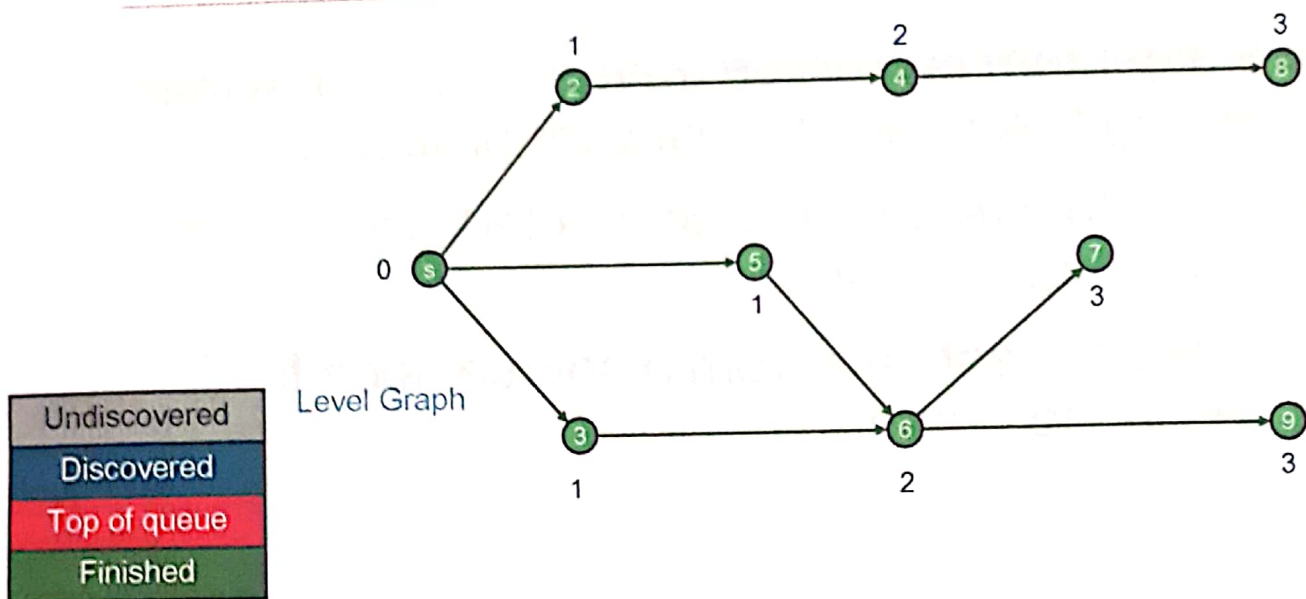
Breadth First Search



Undiscovered
Discovered
Top of queue
Finished

Queue:

Breadth First Search



Algorithms and data structures-2

LECTURE 5-GRAPH SHORTEST PATH ALGORITHM (DIJKSTRA'S ALGORITHM)
ENG.TALAL SULTAN



Dijkstra's Algorithm

- Dijkstra's algorithm is a graph traversal algorithm that solves the single-source shortest path problem.
- It works on both directed and undirected graphs with non-negative edge weights
- provides the shortest path from a source node to all other nodes in the graph.

Steps of Dijkstra's Algorithm

1. Initialize the algorithm by setting the distance of the source node to 0 and all other nodes to infinity.

2. Mark all nodes as unvisited then make source node visited.

3. For each neighboring node of the current node:

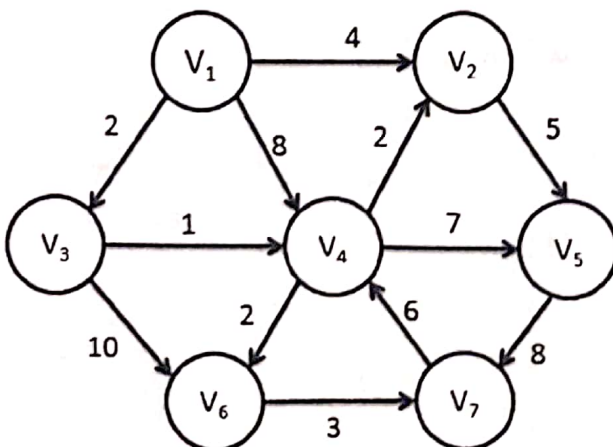
Calculate the distance from the source node to the neighboring node through the current node.

If this calculated distance is smaller than the previously recorded distance for the neighboring node, update it.

Steps of Dijkstra's Algorithm(cont')

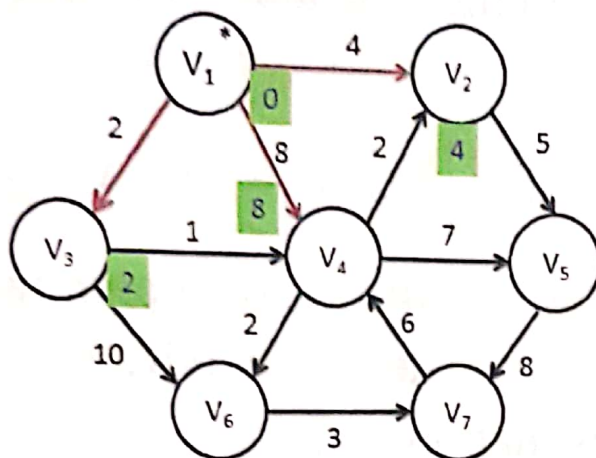
4. Select the node with the smallest distance and mark it as visited.
5. Repeat steps 3 and 4 until all nodes have been visited.
6. Once the algorithm is complete, the shortest path from the source to any other node can be determined by backtracking from the destination node.

Example of Dijkstra's algorithm



V	Known	d_v	p_v
V_1	0	—	0
V_2	0	—	0
V_3	0	—	0
V_4	0	—	0
V_5	0	—	0
V_6	0	—	0
V_7	0	—	0

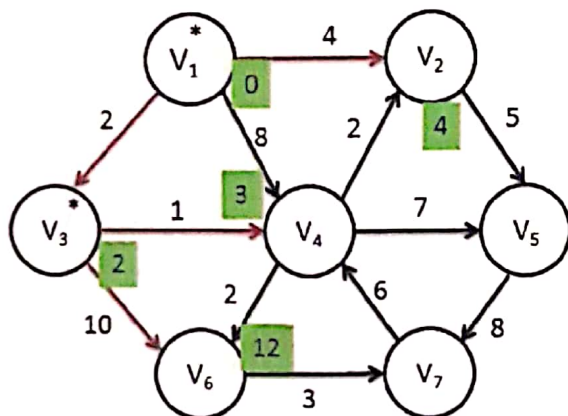
Example of Dijkstra's algorithm



V	Known	d_v	p_v
V_1	1	0	0
V_2	0	4	V_1
V_3	0	2	V_1
V_4	0	8	V_1
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

we select the source vertex as V_1 , with path length 0 and we set known value to 1 and update the distance value of adjacent vertices such as V_2 , V_3 , and V_4

Example of Dijkstra's algorithm



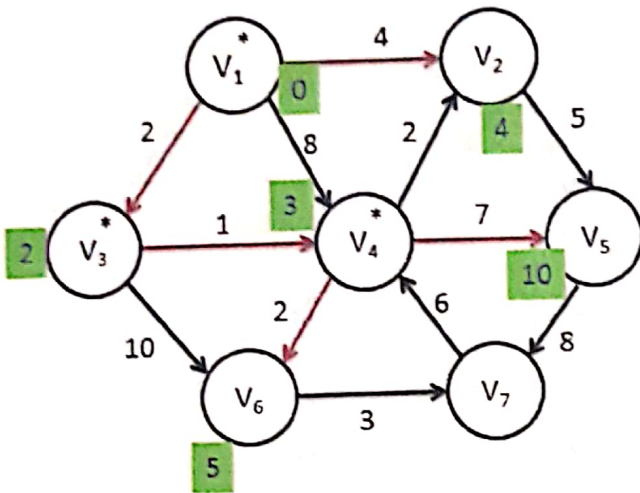
V	Known	d_v	p_v
V_1	1	0	0
V_2	0	4	V_1
V_3	1	2	V_1
V_4	0	3	V_3
V_5	0	∞	0
V_6	0	12	V_3
V_7	0	∞	0

$$d_{V_4} = d_{V_3} + C_{V_3, V_4} \\ = 2 + 1 = 3$$

$$d_{V_6} = d_{V_3} + C_{V_3, V_6} \\ = 2 + 10 = 12$$

Next, V_3 is selected and set known value to 1 and update the adjacent vertices V_4 and V_6

Example of Dijkstra's algorithm



V	Known	d_v	p_v
V_1	1	0	0
V_2	0	4	V_1
V_3	1	2	V_1
V_4	1	3	V_3
V_5	0	10	V_4
V_6	0	5	V_4
V_7	0	∞	0

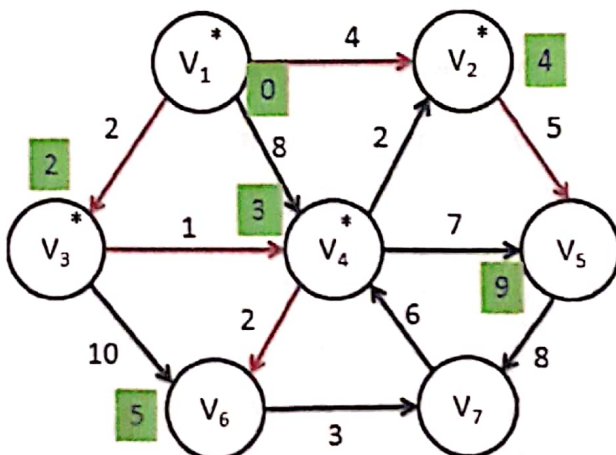
$$d_{V_2} = d_{V_1} + C_{V_1, V_2} = 0 + 4 = 4$$

$$d_{V_5} = d_{V_4} + C_{V_4, V_5} = 3 + 7 = 10$$

$$d_{V_6} = d_{V_4} + C_{V_4, V_6} = 3 + 2 = 5$$

Next, V_4 is selected and marked known. The vertices V_2 , V_5 and V_6 are adjacent.

Example of Dijkstra's algorithm



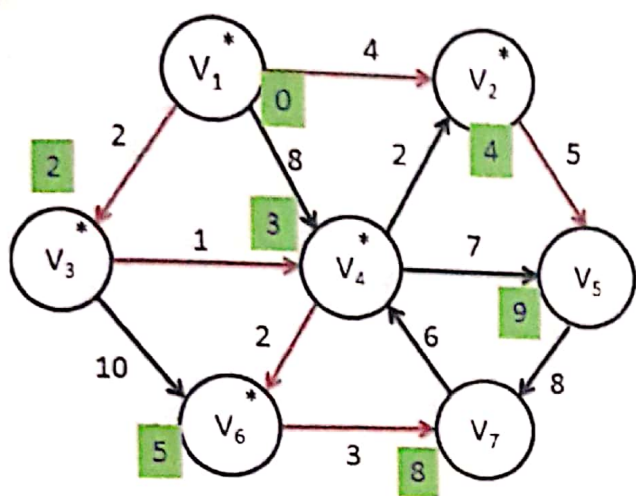
V	Known	d_v	p_v
V_1	1	0	0
V_2	1	4	V_1
V_3	1	2	V_1
V_4	1	3	V_3
V_5	0	9	V_2
V_6	0	5	V_4
V_7	0	∞	0

$$d_{V_5} = d_{V_2} + C_{V_2, V_5} = 4 + 5 = 9$$

V_2 is selected and marked known. V_5 is the only adjacent vertex



Example of Dijkstra's algorithm

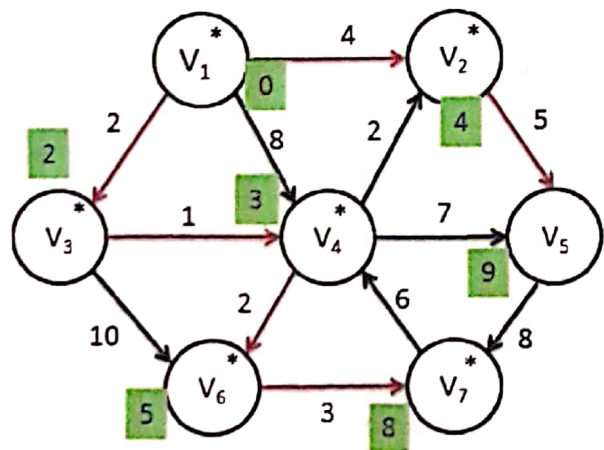


V	Known	d_v	p_v
V_1	1	0	0
V_2	1	4	V_1
V_3	1	2	V_1
V_4	1	3	V_3
V_5	0	9	V_2
V_6	1	5	V_4
V_7	0	8	V_6

$$d_{v7} = d_{v6} + C_{v6, v7}$$

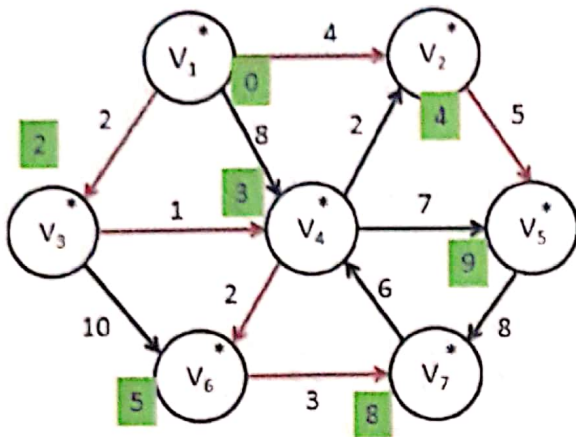
$$= 5 + 3 = 8$$

Example of Dijkstra's algorithm



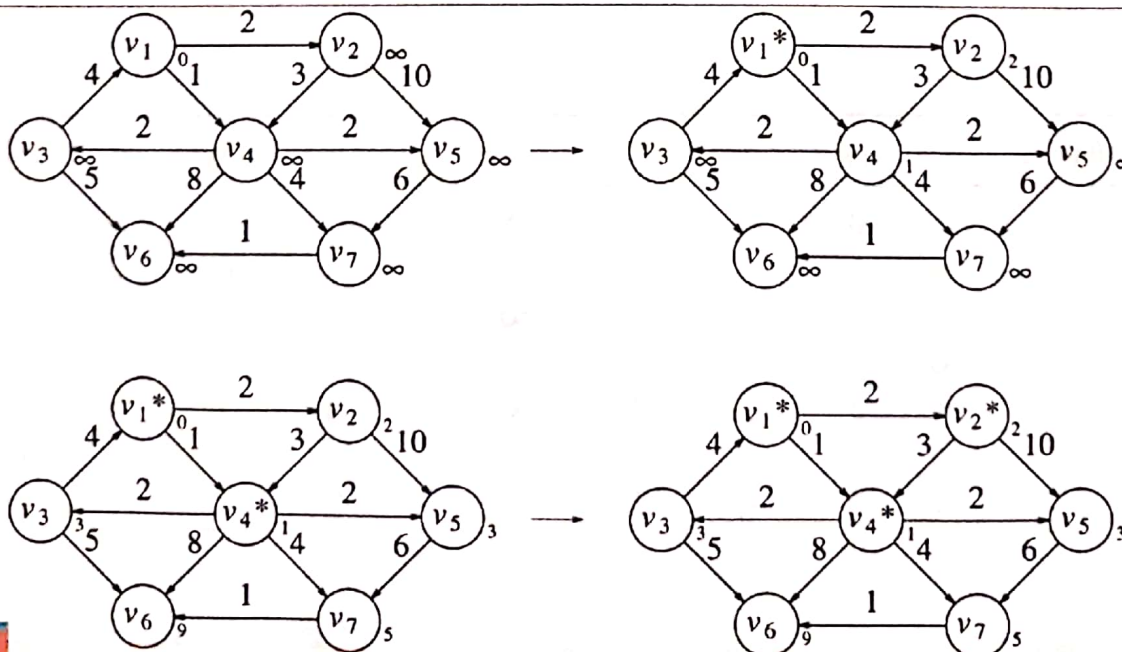
V	Known	d_v	p_v
V_1	1	0	0
V_2	1	4	V_1
V_3	1	2	V_1
V_4	1	3	V_3
V_5	0	9	V_2
V_6	1	5	V_4
V_7	1	8	V_6

Example of Dijkstra's algorithm



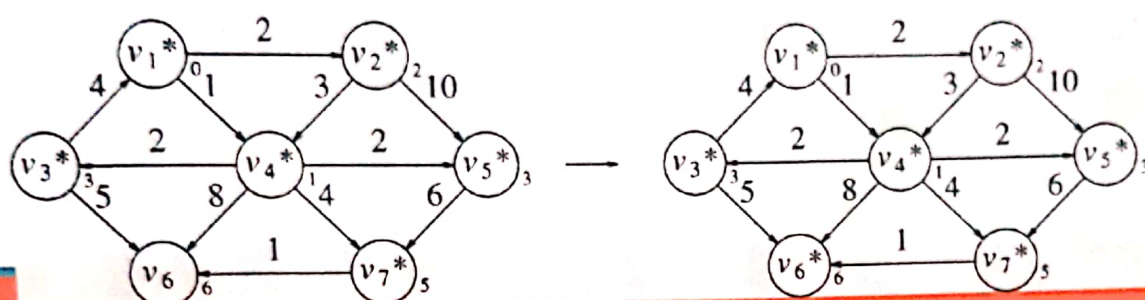
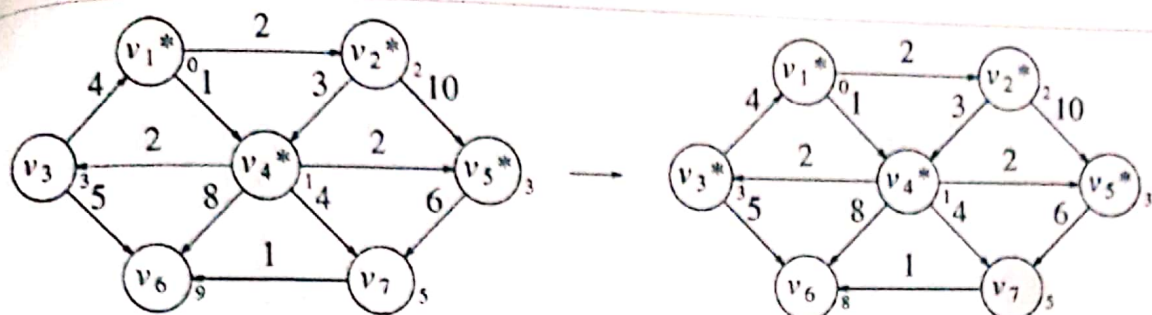
V	Known	d_v	p_v
V_1	1	0	0
V_2	1	4	V_1
V_3	1	2	V_1
V_4	1	3	V_3
V_5	1	9	V_2
V_6	1	5	V_4
V_7	1	8	V_6

Example of Dijkstra's algorithm





Example of Dijkstra's algorithm



AND ALGORITHMS

Example of Dijkstra's algorithm



v	known	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	known	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Example of Dijkstra's algorithm

v	$known$	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

v	$known$	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

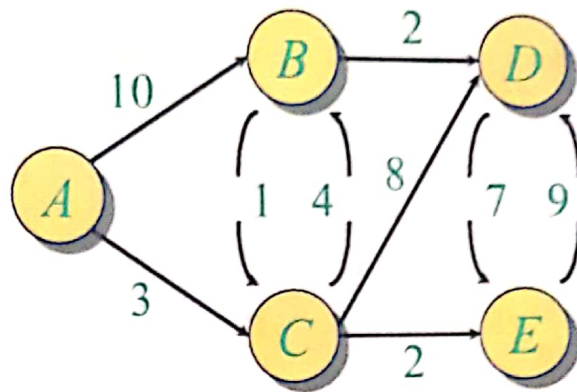
v	$known$	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

Example of Dijkstra's algorithm

v	$known$	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4



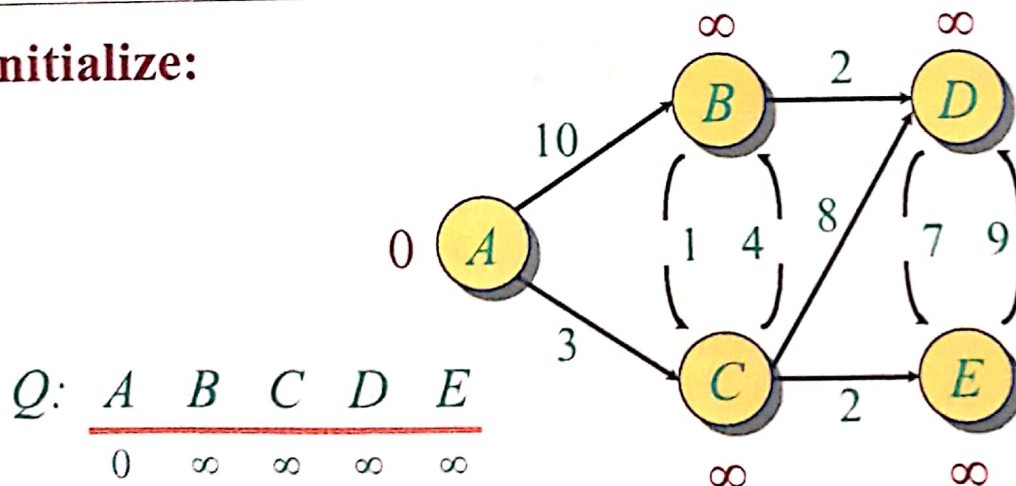
Example of Dijkstra's algorithm



Example of Dijkstra's algorithm



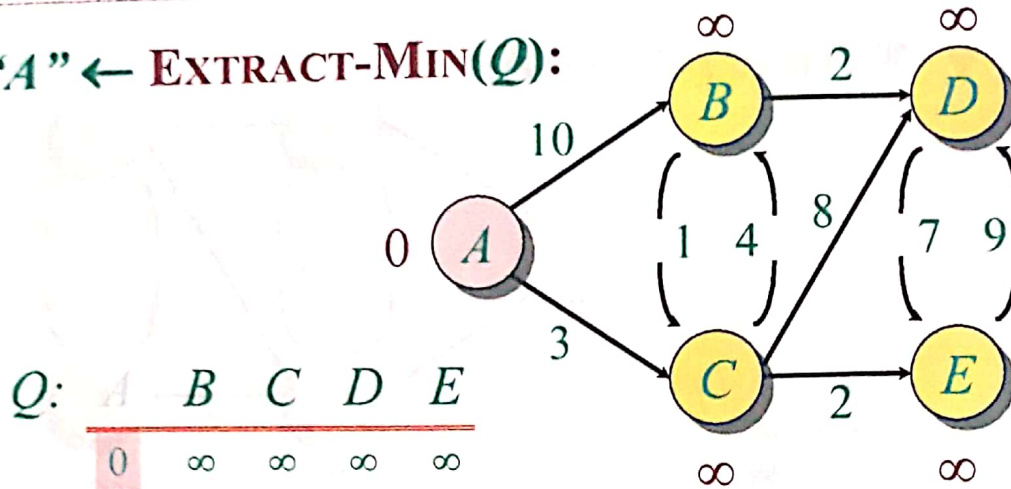
Initialize:



$S: \{\}$

Example of Dijkstra's algorithm

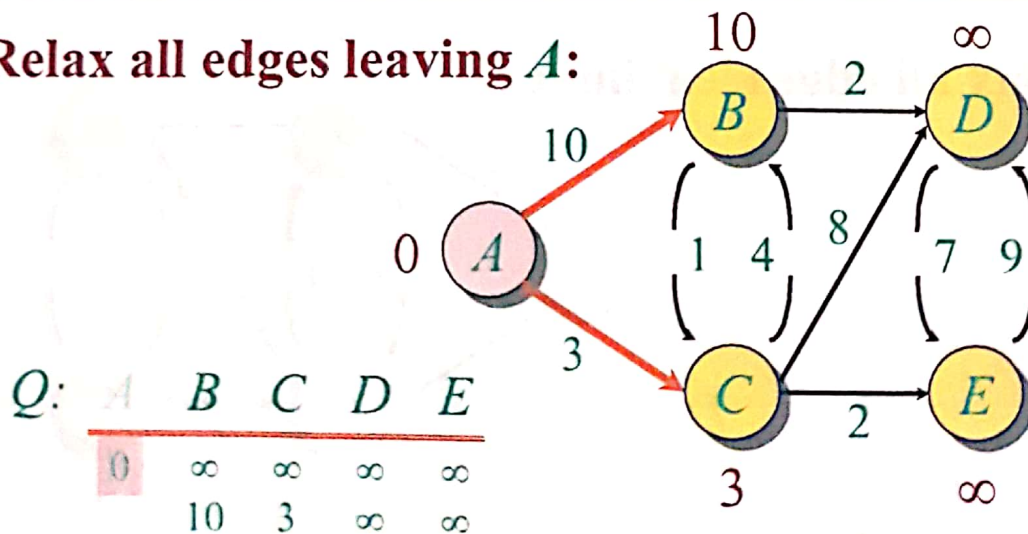
"A" $\leftarrow$ EXTRACT-MIN(Q):



S: { A }

Example of Dijkstra's algorithm

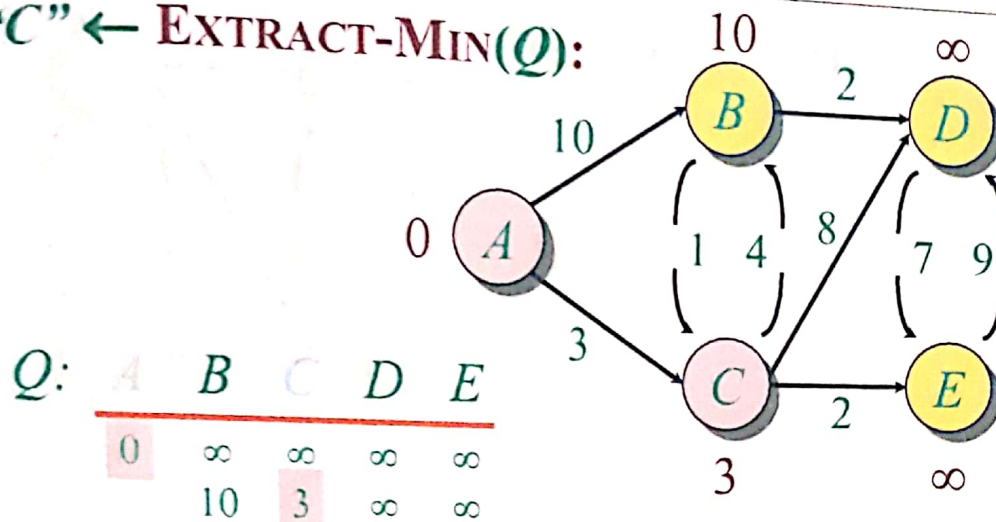
Relax all edges leaving A:



S: { A }

Example of Dijkstra's algorithm

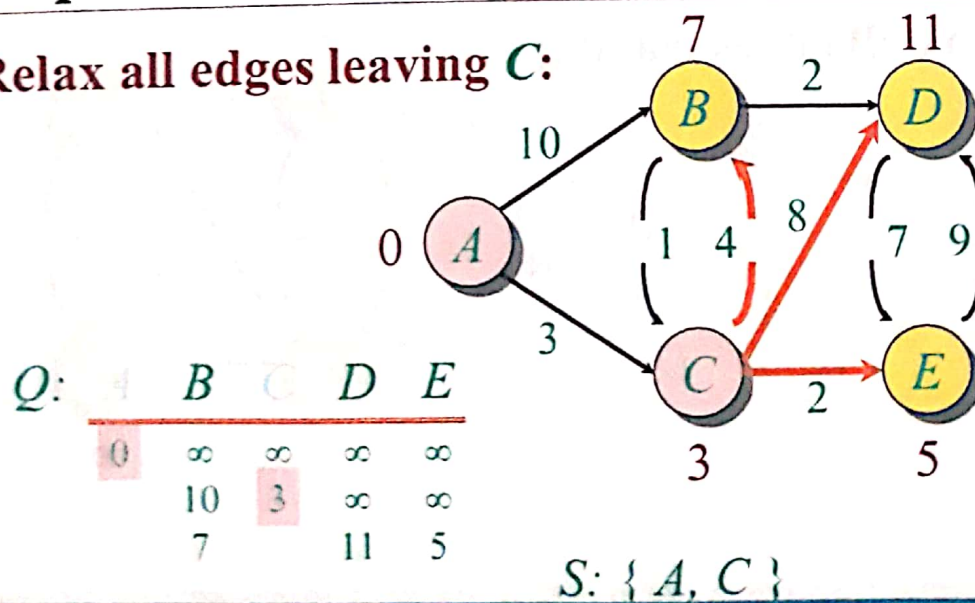
"C" $\leftarrow$ EXTRACT-MIN(Q):



S: {A, C}

Example of Dijkstra's algorithm

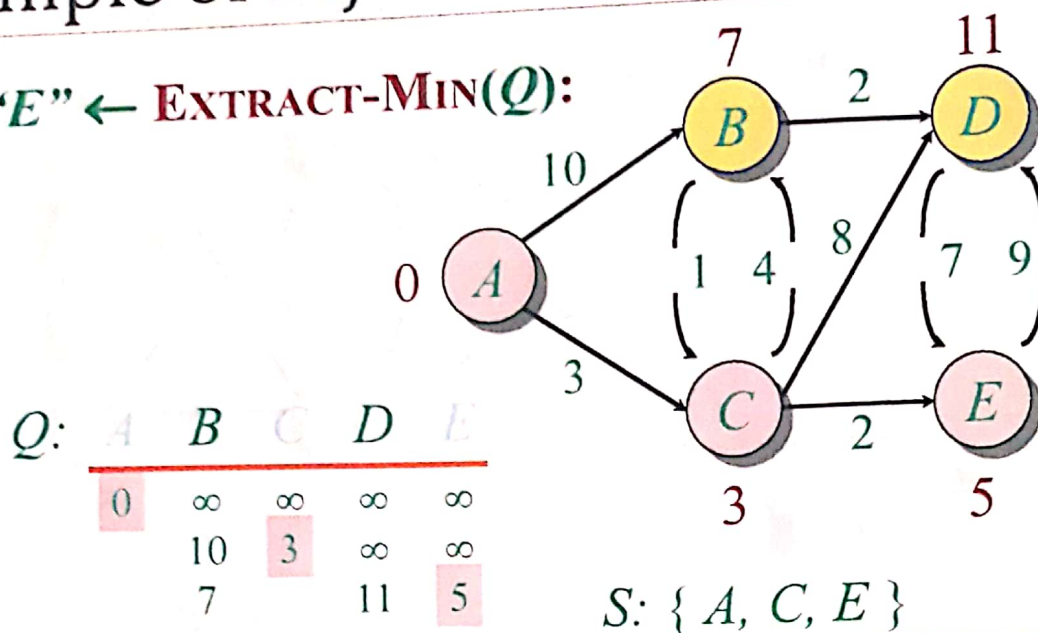
Relax all edges leaving C:



S: {A, C}

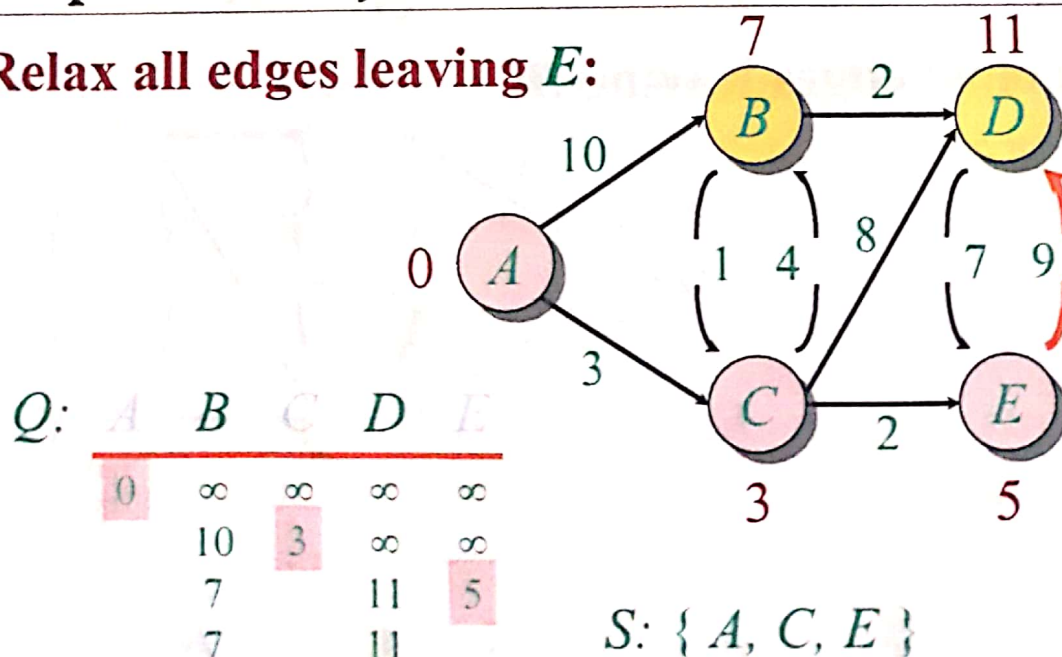
Example of Dijkstra's algorithm

"E" $\leftarrow$ EXTRACT-MIN(Q):



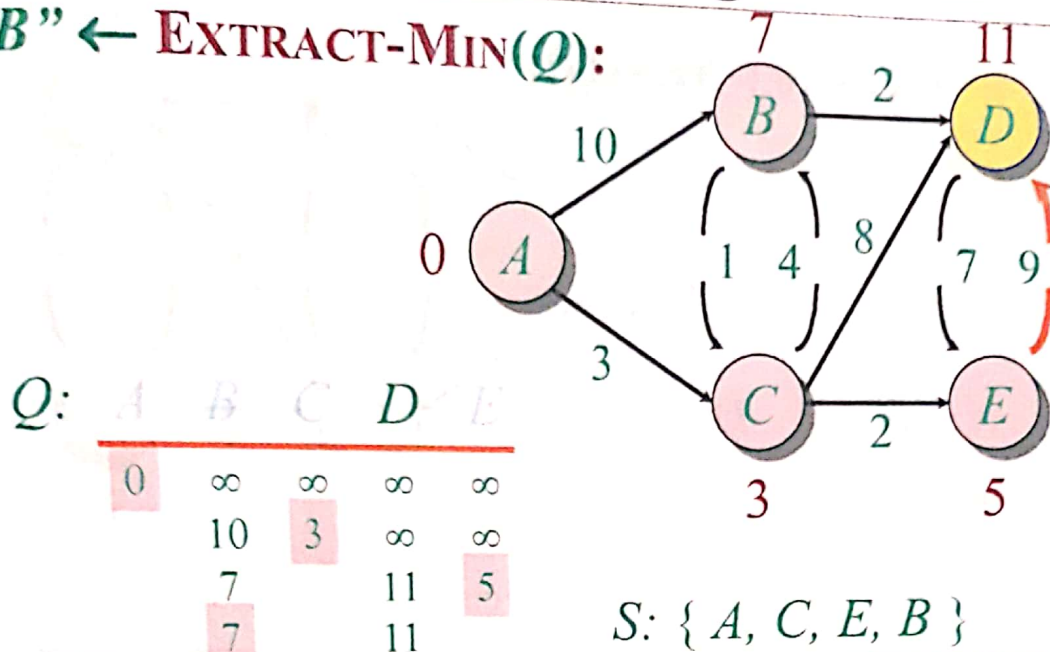
Example of Dijkstra's algorithm

Relax all edges leaving E:



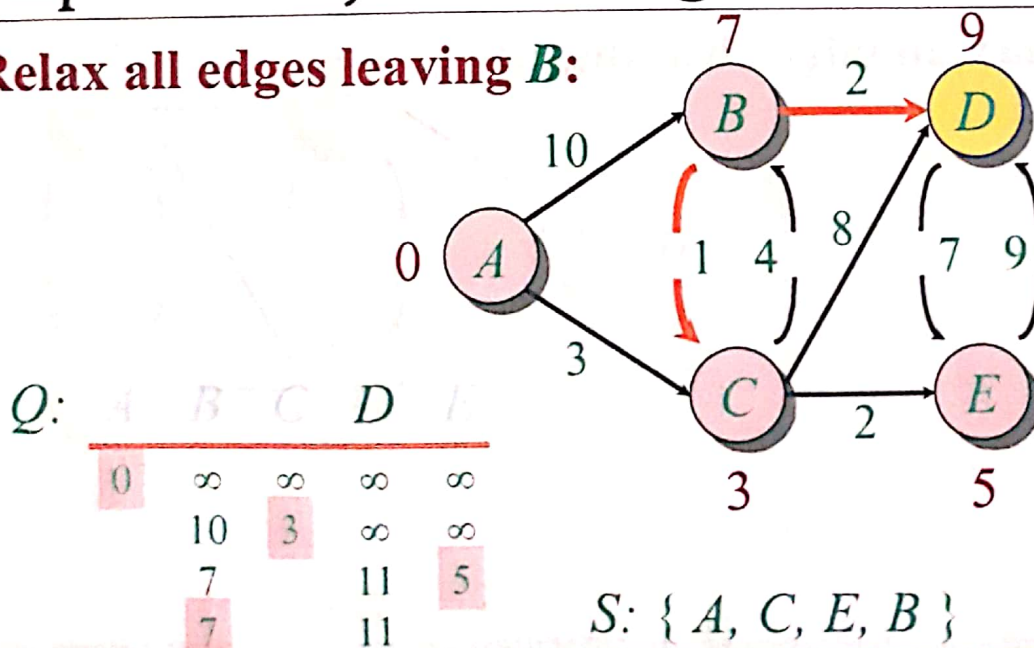
Example of Dijkstra's algorithm

"B" ← EXTRACT-MIN(Q):



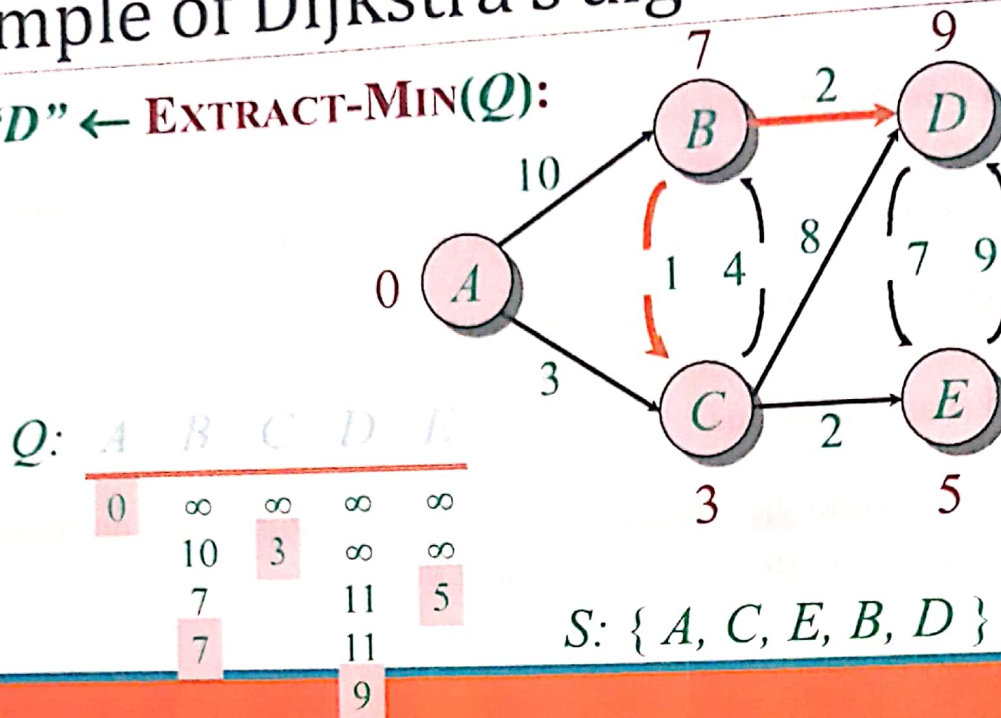
Example of Dijkstra's algorithm

Relax all edges leaving B:



Example of Dijkstra's algorithm

"D" $\leftarrow$ EXTRACT-MIN(Q):



Dijkstra's algorithm

Using an adjacency matrix for Dijkstra's algorithm is suitable when the graph is dense. In such cases, the $O(V^2)$ time complexity is acceptable. However, if the graph is sparse then using an adjacency list implementation would be more efficient, as it can reduce the time complexity to $O((V + E) \log V)$.

Therefore, the choice of implementation (adjacency matrix or adjacency list) depends on the characteristics of the graph and the trade-offs between time complexity and space complexity.





Practical Applications of Dijkstra's Algorithm:



- **Routing in Computer Networks:** Dijkstra's algorithm is extensively used in routing algorithms to find the shortest path between routers or nodes in a network. It helps determine the most efficient path for data packets to traverse.
- **GPS Navigation Systems:** Navigation systems use Dijkstra's algorithm to find the shortest route between a source and a destination, considering factors like distance, traffic, and road conditions.
- **Airline Route Planning:** Airlines utilize Dijkstra's algorithm to optimize flight paths and minimize travel time and fuel consumption.



Algorithms and data structures-2

LECTURE 6-GRAPH SHORTEST PATH ALGORITHM (BELLMAN FORD ALGORITHM)

ENG.TALAL SULTAN

Bellman Ford Algorithm

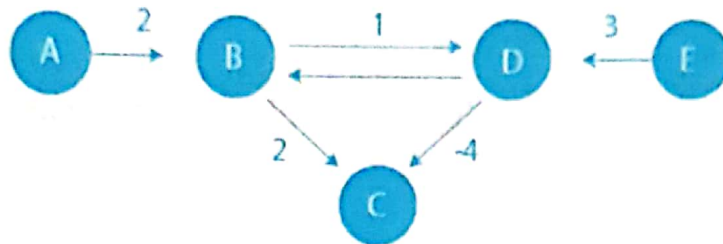
- Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph.
- It is capable of handling graphs with **negative edge** weights, which makes more versatile.
- The shortest path cannot be found if there exists a negative cycle in the graph. If we continue to go around the negative cycle an infinite number of times

Negative weight cycle

- It is important to exercise caution when dealing with negative weight edges in a graph due to the possibility of generating negative weight cycles.
- **Negative weight cycles** are cycles that have the same vertex at the beginning and end, and their total sum of edge weights is negative.
- These cycles pose a challenge for shortest path algorithms as they cannot accurately detect them, leading to incorrect results.



Negative weight cycle



Bellman-Ford Algorithm Steps



- **Step 1** : Initialize the distance to the source vertex as 0, and the distance to all other vertices as infinity.
- **Step 2** : **Relax all edges** in the graph $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. For each edge (u, v) with weight w , check if the distance from the source vertex to v can be reduced by going through u . If so, update the distance to v to the new, shorter distance.

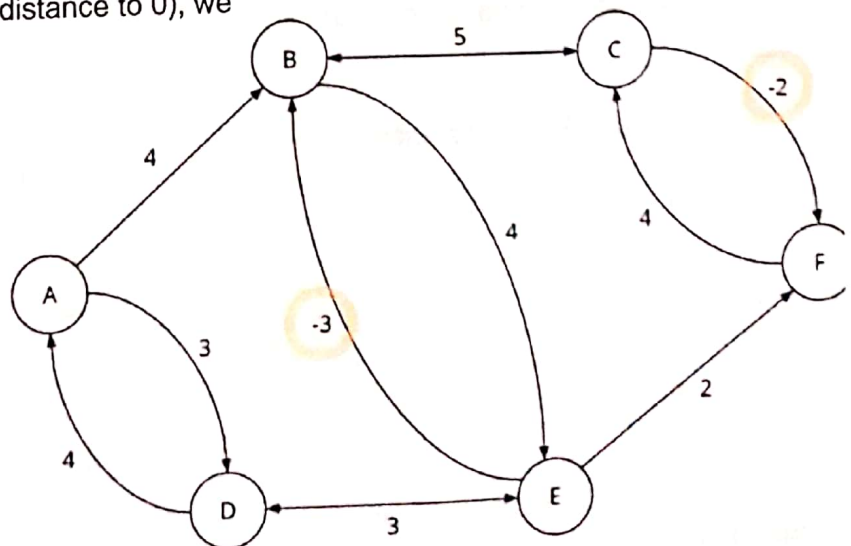
Bellman-Ford Algorithm Steps

- **Step 3** : Check for negative weight cycles. If there is a negative weight cycle in the graph, the algorithm will never converge and will keep reducing the distance to some vertices with each iteration. To detect such cycles, repeat step 2 one more time. **If any distance is updated in this extra iteration, there must be a negative weight cycle in the graph.**
- **Step 4** : If there is no negative weight cycle, the shortest distance to each vertex from the source vertex has been found

Example of Bellman-Ford

Using vertex A as the source (setting its distance to 0), we initialize all the other distances to ∞ .

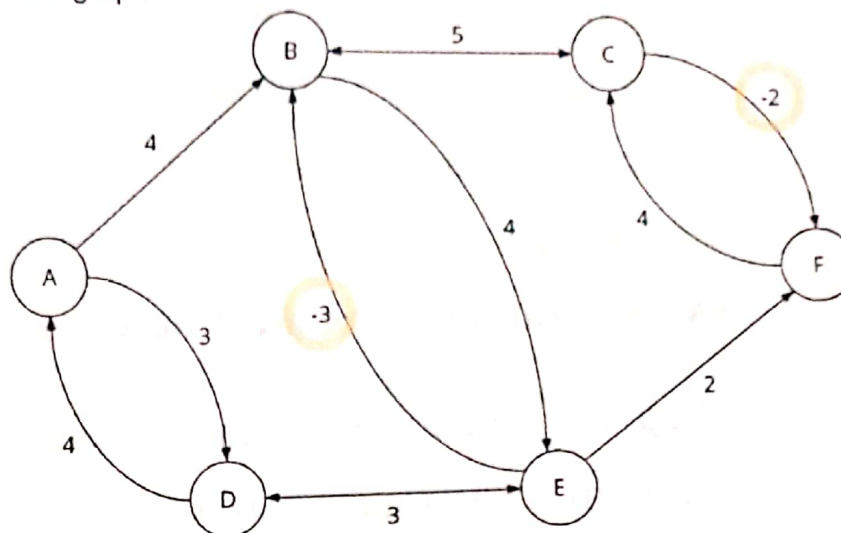
Node	P	distance
A	-	0
B	-	∞
C	-	∞
D	-	∞
E	-	∞
F	-	∞



Iteration 1 of 5

In each iteration, we examine *all* edges of the graph start with the edge (a, b):

Node	P	distance
A	-	0
B	A	4
C	-	∞
D	-	∞
E	-	∞
F	-	∞

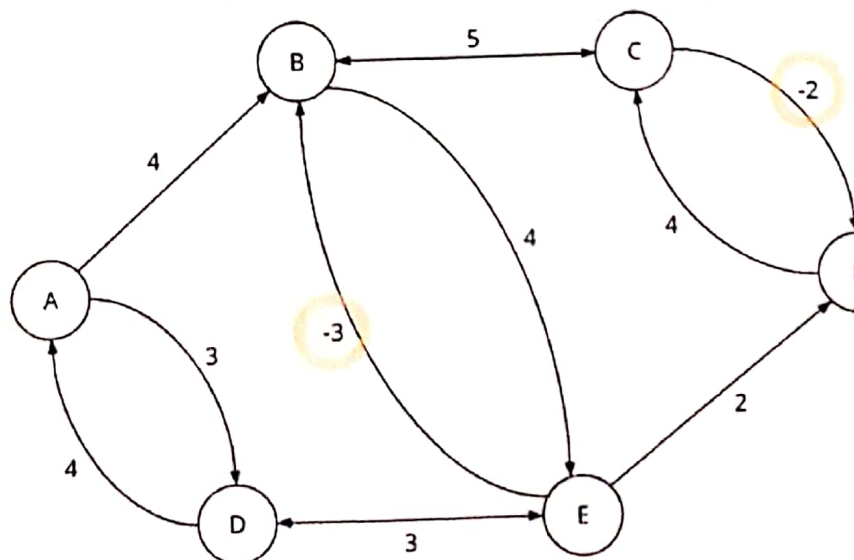


edge (a, b) 0 (total cost from start to A)
 + 4 (cost A→B)
 = 4

Iteration 1 of 5

We next examine edge (a, d):

Node	P	distance
A	-	0
B	A	4
C	-	∞
D	A	3
E	-	∞
F	-	∞



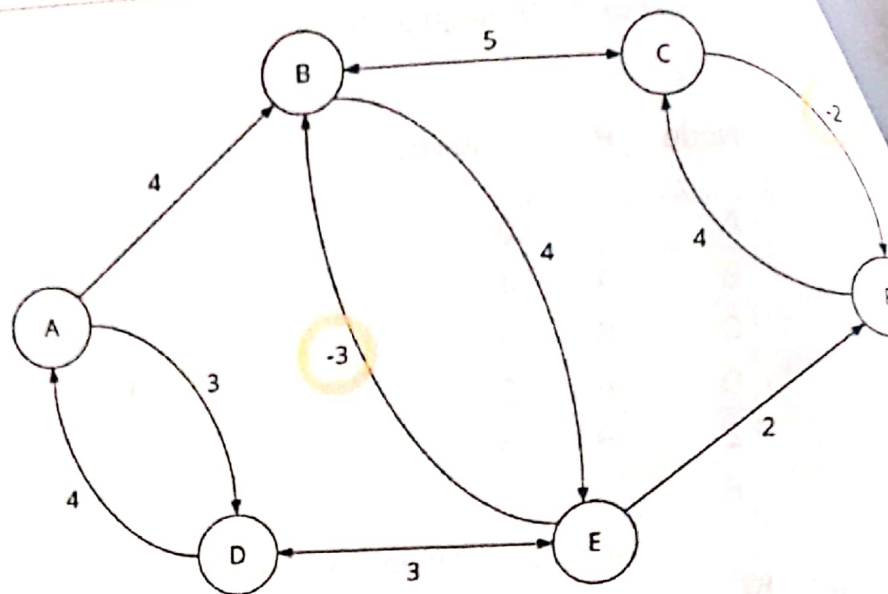
Edge (a, d) 0 (total cost from start to A)
 + 3 (cost A→D)
 = 3

Iteration 1 of 5

We next examine **edge (b, c)**

Node	P	distance
A	-	0
B	A	4
C	B	9
D	A	3
E	-	∞
F	-	∞

Edge (b, c) 4 (total cost from start to B)
 + 5 (cost B→C)
 = 9

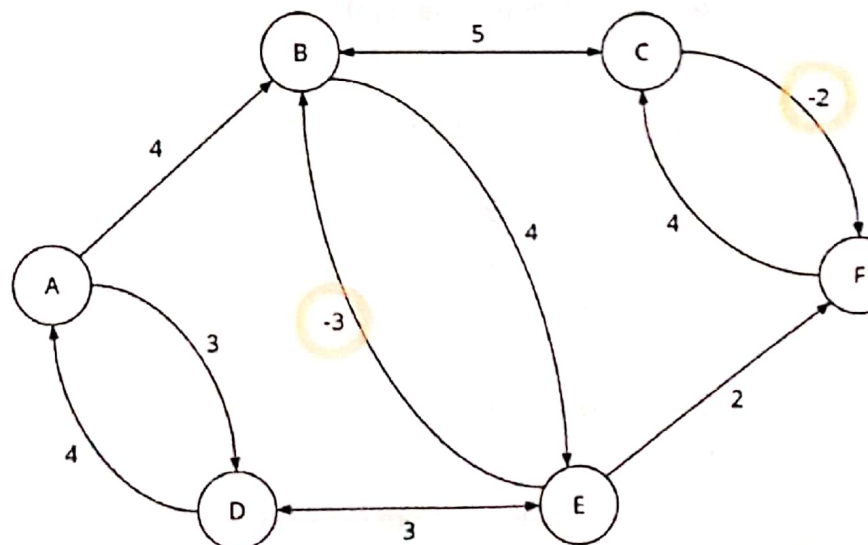


Iteration 1 of 5

We next examine **edge (b, e)**

Node	P	distance
A	-	0
B	A	4
C	B	9
D	A	3
E	B	8
F	-	∞

Edge (b, e) 4 (total cost from start to B)
 + 4 (cost B→E)
 = 8 < ∞ update E

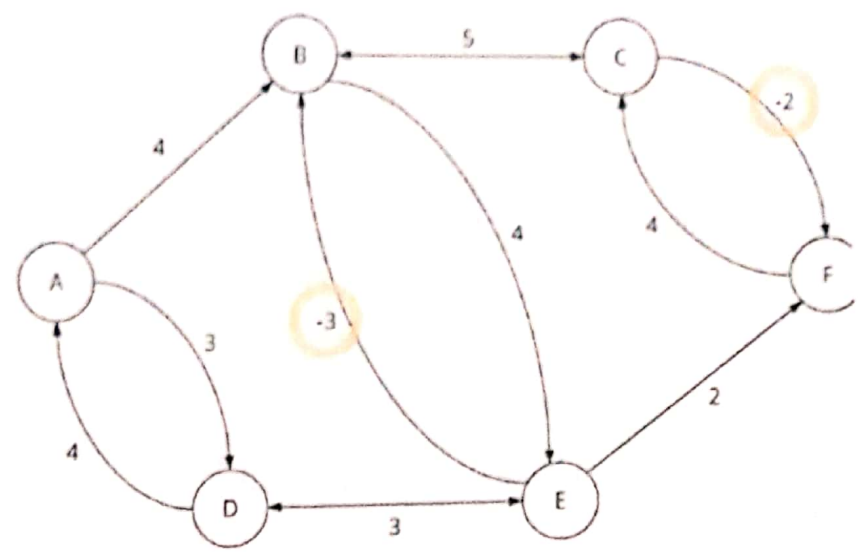


Iteration 1 of 5



We next examine edge (c, b)

Node	P	distance
A	-	0
B	A	4
C	B	9
D	A	3
E	B	8
F	-	∞



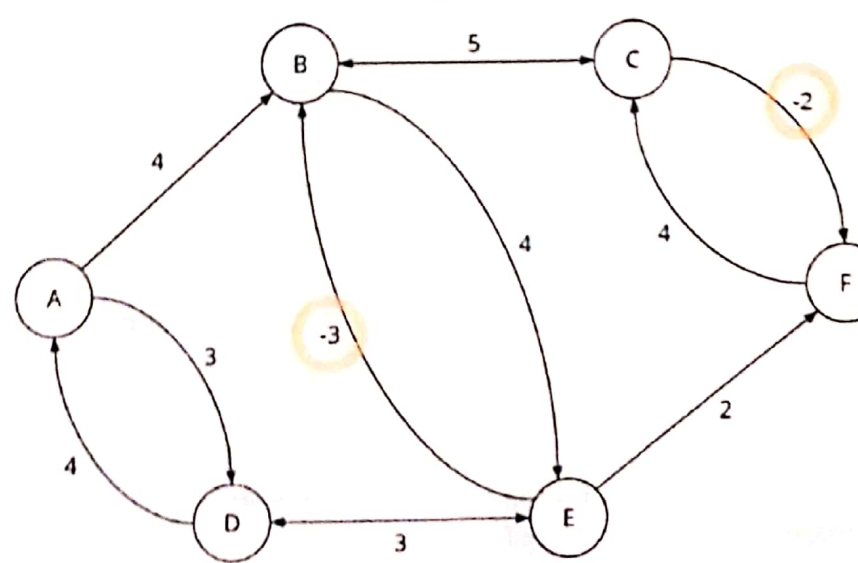
Edge (c, b) 9 (total cost from start to C)
 + 5 (cost C→B)
 = 14 don't update B

Iteration 1 of 5



We next examine edge (c, f)

Node	P	distance
A	-	0
B	A	4
C	B	9
D	A	3
E	B	8
F	C	7

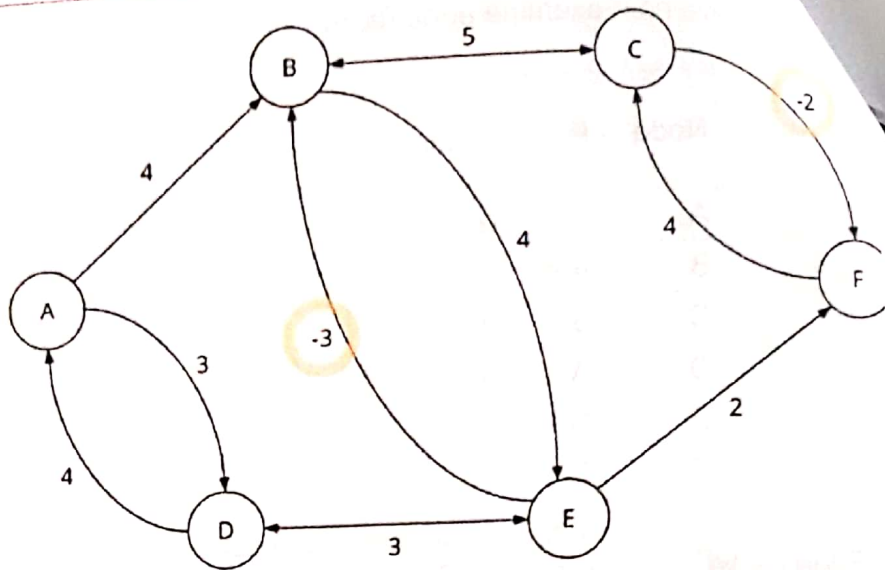


Edge (c, f) 9 (total cost from start to C)
 - 2 (cost C→F)
 = 7 < ∞ update F

Iteration 1 of 5

We next examine edge (d, a)

Node	P	distance
A	-	0
B	A	4
C	B	9
D	A	3
E	B	8
F	C	7



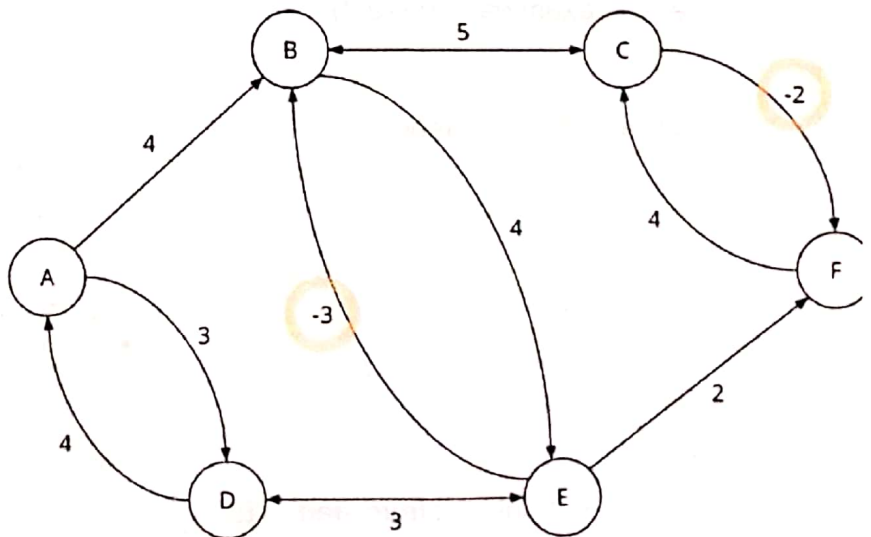
Edge (d, a) 3 (total cost from start to D)
+ 4 (cost D→A)
= 7 > 0 don't update A



Iteration 1 of 5

We next examine edge (d, e)

Node	P	distance
A	-	0
B	A	4
C	B	9
D	A	3
E	B D	8 6
F	C	7



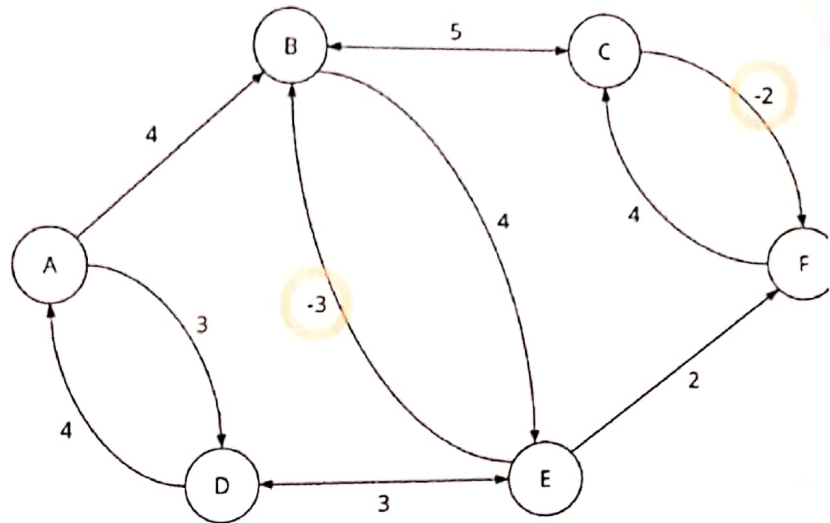
Edge (d, e) 3 (total cost from start to D)
+ 3 (cost D→E)
= 6 < 8 update E

Iteration 1 of 5



We next examine edge (e, b)

Node	P	distance
A	-	0
B	A-E	4-3
C	B	9
D	A	3
E	D	6
F	C	7



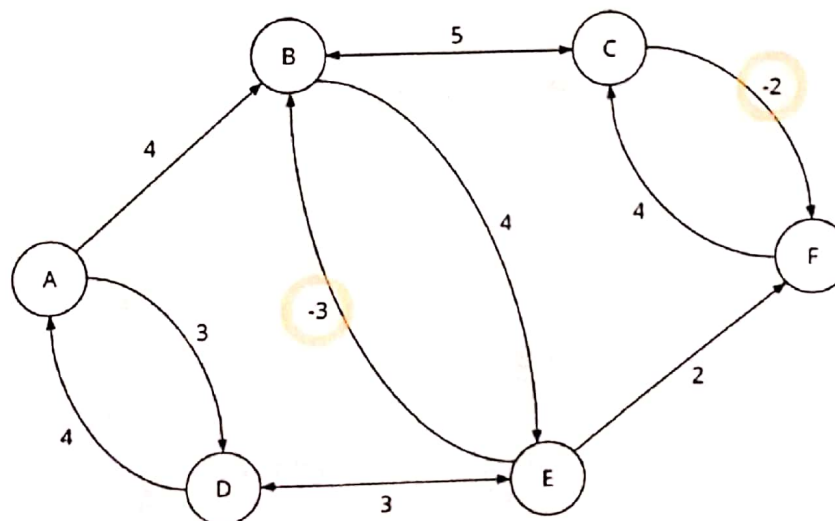
6 (total cost from start to E)
 Edge (e, b) - 3 (cost E→B)
 = 3 < 4 Update B

Iteration 1 of 5



We next examine edge (e, f)

Node	P	distance
A	-	0
B	E	3
C	B	9
D	A	3
E	D	6
F	C	7



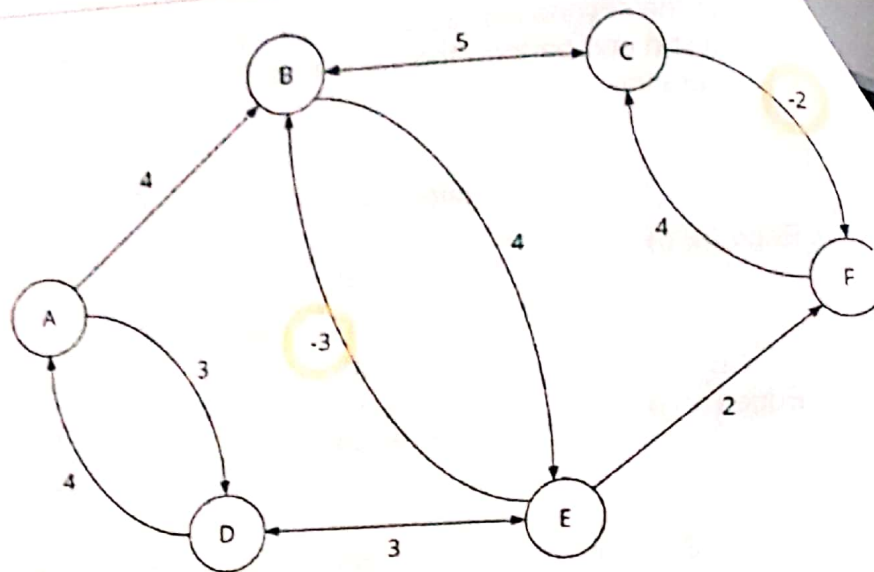
6 (total cost from start to E)
 Edge (e, f) + 2 (cost E→F)
 = 8 > 7 don't update

Iteration 1 of 5

We next examine edge (f, c)

Node	P	distance
A	-	0
B	E	3
C	B	9
D	A	3
E	D	6
F	C	7

Edge (f, c)
 7 (total cost from start to F)
 + 4 (cost F→C)
 = 11 > 9 don't update



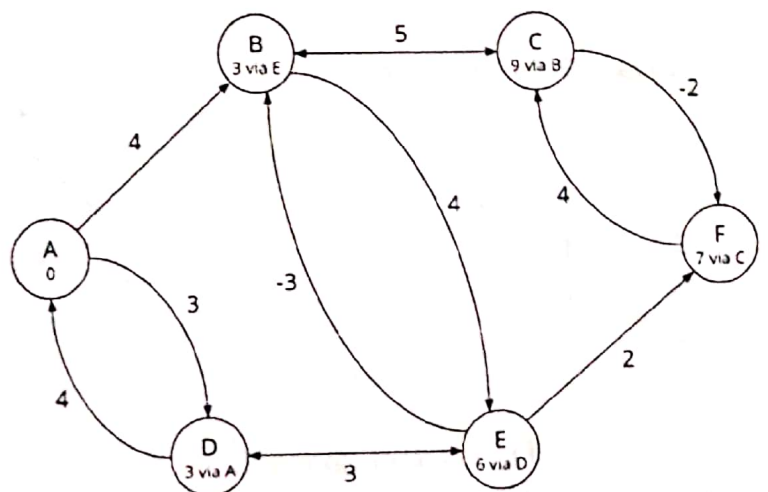
Iteration 1 of 5



End of the First Iteration

We have now examined all edges of the graph exactly once

Node	P	distance
A	-	0
B	E	3
C	B	9
D	A	3
E	D	6
F	C	7





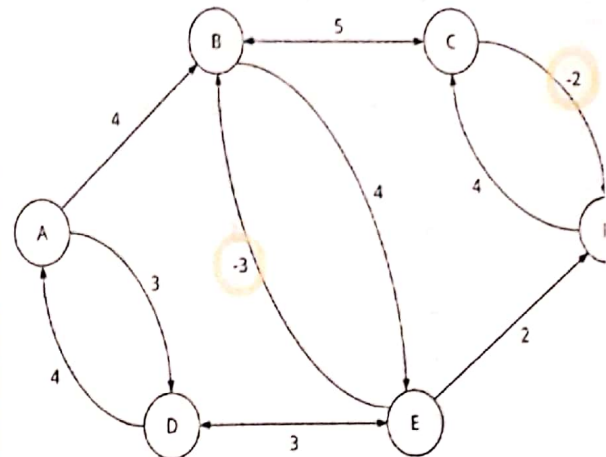
Iteration 2 of 5

In the second iteration, we examine all the graph's edges again and perform the same calculations as in the first iteration.

Edge (a, b) 0 (total cost from start to A)
+ 4 (cost A→B)
= 4

Edge (a, d) 0 (total cost from start to A)
+ 3 (cost A→D)
= 3

Node	P	distance
A	—	0
B	E	3
C	B	9
D	A	3
E	D	6
F	C	7



Since the total cost of node A did not change in the previous iteration

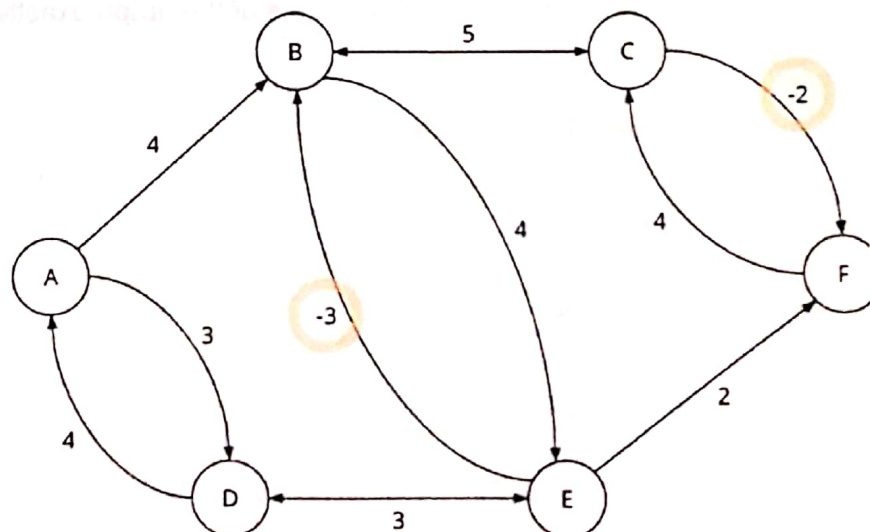
Iteration 2 of 5



We next examine edge **Edge (b, c)**

Node	P	distance
A	—	0
B	E	3
C	B	8
D	A	3
E	D	6
F	C	7

Edge (b, c) 3 (total cost from start to B)
+ 5 (cost B→C)
= 8 < 9



Iteration 2 of 5

Edges (b, e) and (c, b)

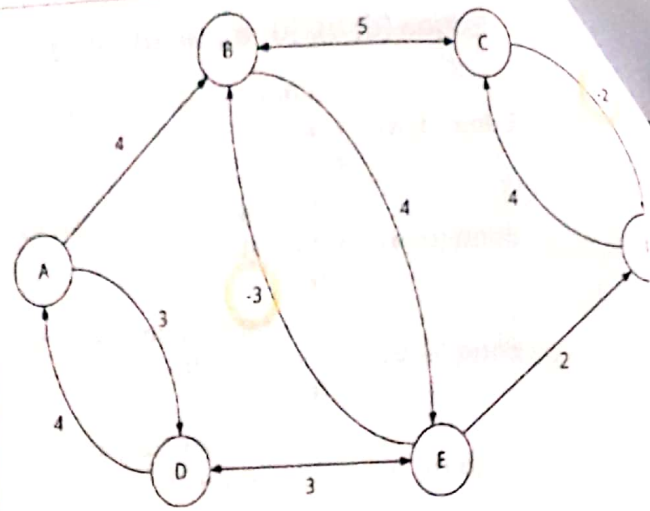
Edge (b, e)

3 (total cost from start to B)
+ 4 (cost B→E)
= 7

Edge (c, b)

8 (total cost from start to C)
+ 5 (cost C→B)
= 13

Node	P	distance
A	-	0
B	E	3
C	B	8
D	A	3
E	D	6
F	C	7



In both cases, the edge end node's total cost is higher than currently stored, don't update

Iteration 2 of 5

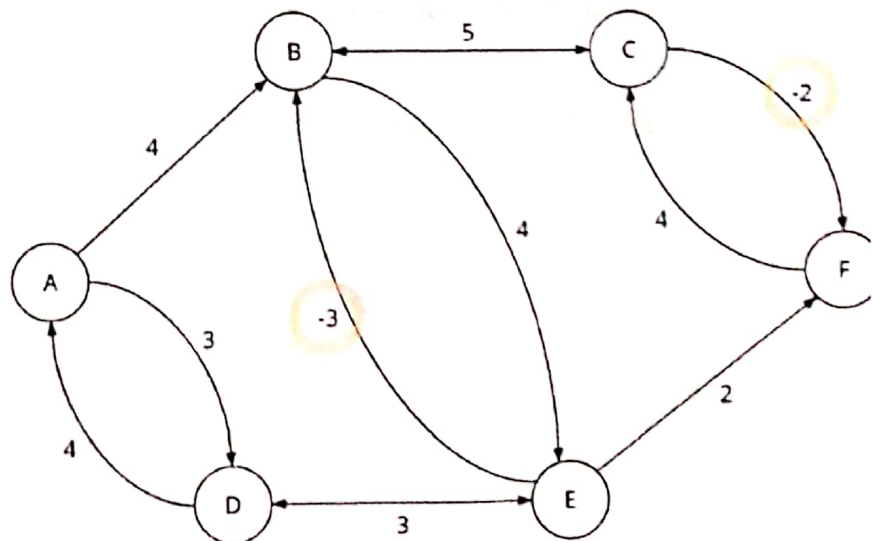


We next examine edge **Edge (c, f)**

Node	P	distance
A	-	0
B	E	3
C	B	8
D	A	3
E	D	6
F	C	6

Edge (c, f)

8 (total cost from start to C)
- 2 (cost C→F)
= 6 < 7 update F





Iteration 2 of 5



Edges (d, a), (d, e), (e, b), (e, f), and (f, c)

Edge (d, a) 3 (total cost from start to D)
+ 4 (cost D→A)
= 7

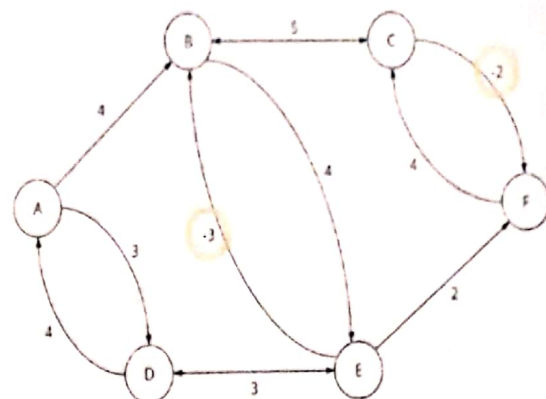
Edge (d, e) 3 (total cost from start to D)
+ 3 (cost D→E)
= 6

Edge (e, b) 6 (total cost from start to E)
- 3 (cost E→B)
= 3

Edge (e, f) 6 (total cost from start to E)
+ 2 (cost E→F)
= 8

Edge (f, c) 6 (total cost from start to F)
+ 4 (cost F→C)
= 10

Node	P	d
A	-	0
B	E	3
C	B	8
D	A	3
E	D	6
F	C	6



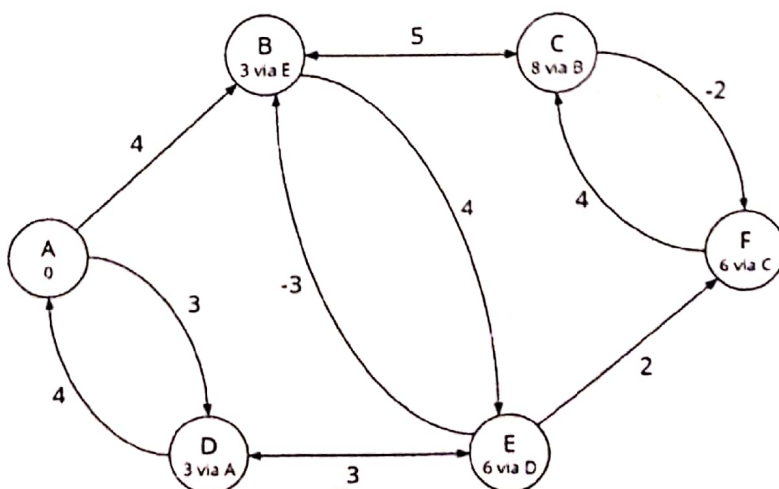
The newly calculated total cost for the edge end node is greater than or equal to the current value in all five cases. Thus, there are no further changes.

Iteration 2 of 5



End of the Second Iteration

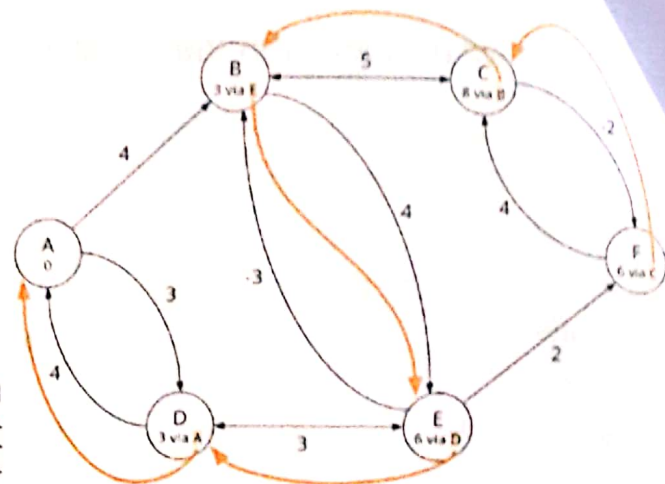
Node	P	distance
A	-	0
B	E	3
C	B	8
D	A	3
E	D	6
F	C	6



Iteration 3 of 5

Node	P	distance
A	-	0
B	E	3
C	B	8
D	A	3
E	D	6
F	C	6

After the third check of all edges, the algorithm will not have detected any further cost reductions. therefore terminate prematurely at the end of iteration 3. and don't need to continue iterations

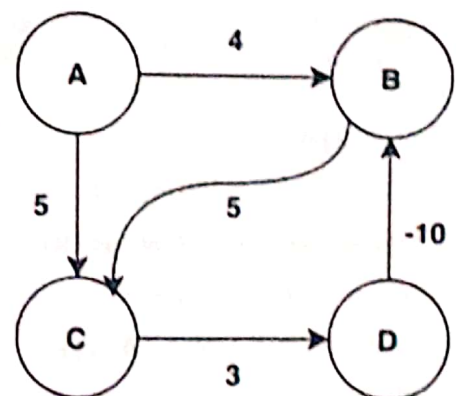


Short path from A to F

Example of Bellman-Ford (negative cycle)



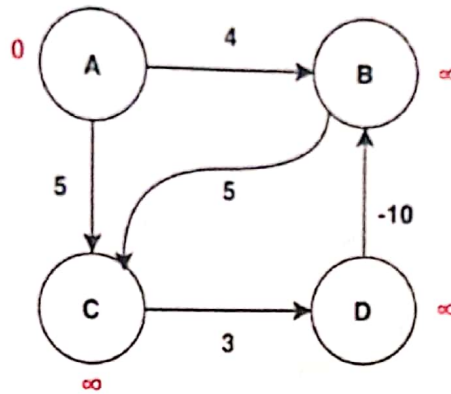
The graph has 4 vertices. We're considering the vertex A as the starting vertex here. The algorithm expects to iterate 3 times for calculation of shortest distance and one more time to check for the negative cycle. The edge order we're going to follow here is: (D, B) -> (C, D) -> (A, C) -> (A, B) -> (B, C)



Example of Bellman-Ford (negative cycle)



As we're done with the initialization, we can proceed to the first iteration:



Example of Bellman-Ford (negative cycle)



After the first iteration, we can see changes in the distance value of B and C. From the vertex A, the edge weights are directly assigned to vertex B and C to update their distance value.

For edge (A, C):

$$D[C] > D[A] + W[A, C] \quad \infty > 0 + 5$$

Therefore, the new value of the vertex C is updated:

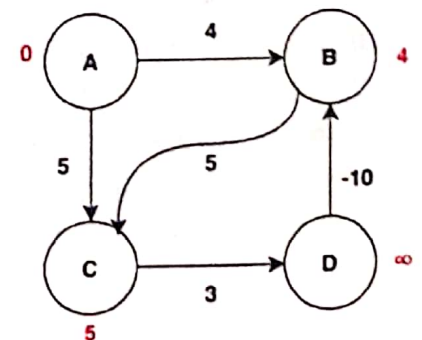
$$D[C] = D[A] + W[A, C] = 0 + 5 = 5$$

Similarly for edge (A, B):

$$D[B] > D[A] + W[A, B] \quad \infty > 0 + 4$$

The algorithm stores the new value of B:

$$D[B] = D[A] + W[A, B] = 0 + 4 = 4$$



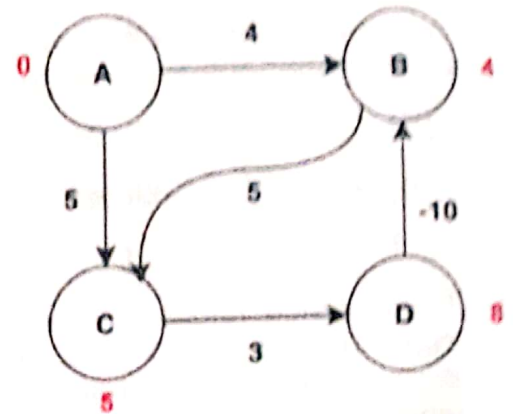
Example of Bellman-Ford (negative cycle)

After the second iteration, the distance value of the vertex D is updated by the algorithm by relaxing the edge (C, D):

$$D[D] > D[C] + W[C, D] \infty > 5 + 3$$

Therefore, the new value of the vertex C is updated:

$$D[D] = D[C] + W[C, D] = 5 + 3 = 8$$



Example of Bellman-Ford (negative cycle)



After third iteration, the values are again getting changed:

Here there are two updates. One for the vertex B and another for vertex C. The update for vertex B is done by relaxing the edge (D, B):

$$D[B] > D[D] + W[D, B] \quad 4 > 8 + (-10) \text{ Therefore,}$$

the new value of the vertex B is updated:

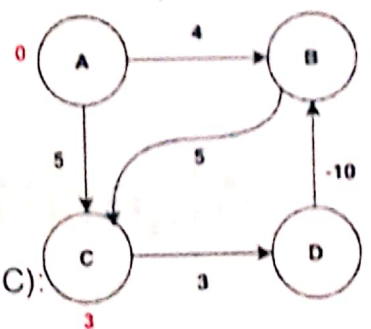
$$D[B] = D[D] + W[D, B] = 8 + (-10) = -2$$

The algorithm updated the value for vertex B by relaxing the edge (B, C):

$$D[C] > D[B] + W[B, C] \quad 5 > (-2) + 5$$

The value of C is updated and stored:

$$D[C] = D[B] + W[B, C] = (-2) + 5 = 3$$



Example of Bellman-Ford (negative cycle)



Now let's iterate one last time to decide whether the graph has a negative cycle or not

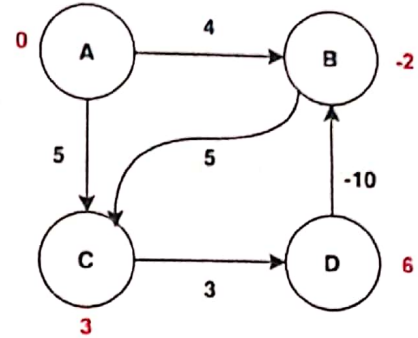
We can see there is a change in value for vertex D. The change happens as the algorithm relax the edge (C, D):

$$D[D] > D[C] + W[C, D] \quad 8 > 3 + 3$$

The value of C is updated and stored:

$$D[C] = D[C] + W[C, D] = 3 + 3 = 6$$

The distance values are not stable even after the maximum number of iterations. Therefore, in this case, the algorithms return that the graph contains a negative weighted cycle, and hence it is not possible to calculate the shortest path from the starting vertex to all other vertices in the given graph.



Algorithms and data structures-2

LECTURE 8-MINIMUM SPANNING TREES

ENG.TALAL SULTAN



Lecture Outline

- Search problem
- Direct Addressing
- Hashing
- Collisions
- Handling Collisions

The Search Problem

Find items with **keys** matching a given **search key**

- Given an array A , containing n keys, and a search key x , find the index i such as $x=A[i]$
- As in the case of sorting, a key could be part of a large record.

example of a record

Key	other data
-----	------------



Applications

Keeping track of customer account information at a bank

- Search through records to check balances and perform transactions

Keep track of reservations on flights

- Search to find empty seats, cancel/modify reservations

Search engine

- Looks for all documents containing a given word



Special Case: Dictionaries

Dictionary = data structure that supports mainly two basic operations: **insert** a new item and **return an item with a given key**

Queries: return information about the set S:

- Search (S, k)

Modifying operations: change the set

- Insert (S, k)
- Delete (S, k)

Direct Addressing

Assumptions:

- Key values are distinct
- Each key is drawn from a universe $U = \{0, 1, \dots, m - 1\}$

Idea:

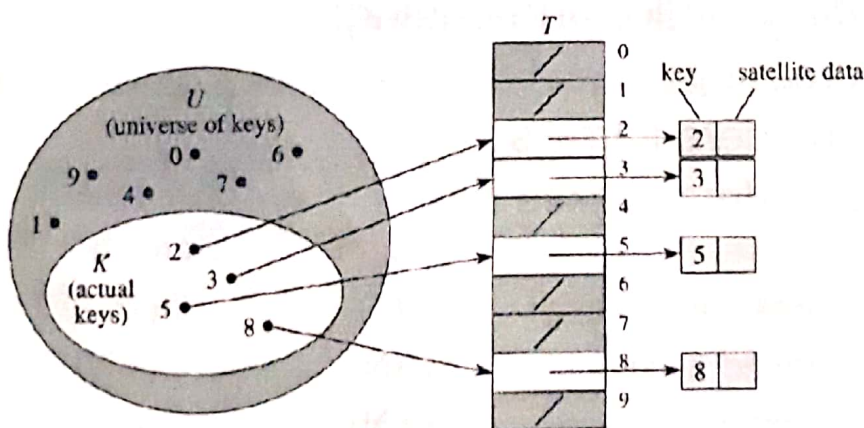
- Store the items in an array, indexed by keys

- **Direct-address table representation:**

- An array $T[0 \dots m - 1]$
- Each **slot**, or position, in T corresponds to a key in U
- For an element x with key k , a pointer to x (or x itself) will be placed in location $T[k]$
- If there are no elements with key k in the set, $T[k]$ is empty, represented by NIL



Direct Addressing (cont'd)



(insert/delete in $O(1)$ time)



Operations

Alg.: DIRECT-ADDRESS-SEARCH(T, k)
 return $T[k]$

Alg.: DIRECT-ADDRESS-INSERT(T, x)
 $T[\text{key}[x]] \leftarrow x$

Alg.: DIRECT-ADDRESS-DELETE(T, x)
 $T[\text{key}[x]] \leftarrow \text{NIL}$

Running time for these operations: $O(1)$

8



Comparing Different Implementations

Implementing dictionaries using:

- Direct addressing
- Ordered/unordered arrays
- Ordered/unordered linked lists

	Insert	Search
direct addressing	$O(1)$	$O(1)$
ordered array	$O(N)$	$O(\lg N)$
ordered list	$O(N)$	$O(N)$
unordered array	$O(1)$	$O(N)$
unordered list	$O(1)$	$O(N)$

9

Examples Using Direct Addressing

Example 1:

- (i) Suppose that the keys are integers from 1 to 100 and that there are about 100 records
- (ii) Create an array A of 100 items and store the record whose key is equal to i in $A[i]$

Example 2:

- (i) Suppose that the keys are nine-digit social security numbers
- (ii) We can use the same strategy as before but it is very inefficient now: an array of 1 billion items is needed to store 100 records !!

- $|U|$ can be very large

- $|K|$ can be much smaller than $|U|$

Hash Tables



When K is much smaller than U , a **hash table** requires much less space than a **direct-address table**

- Can reduce storage requirements to $|K|$
- Can still get $O(1)$ search time, but on the average case, not the worst case

Hash Tables

Idea:

- Use a function h to compute the slot for each key
- Store the element in slot $h(k)$

A **hash function** h transforms a key into an index in a hash table $T[0...m-1]$:

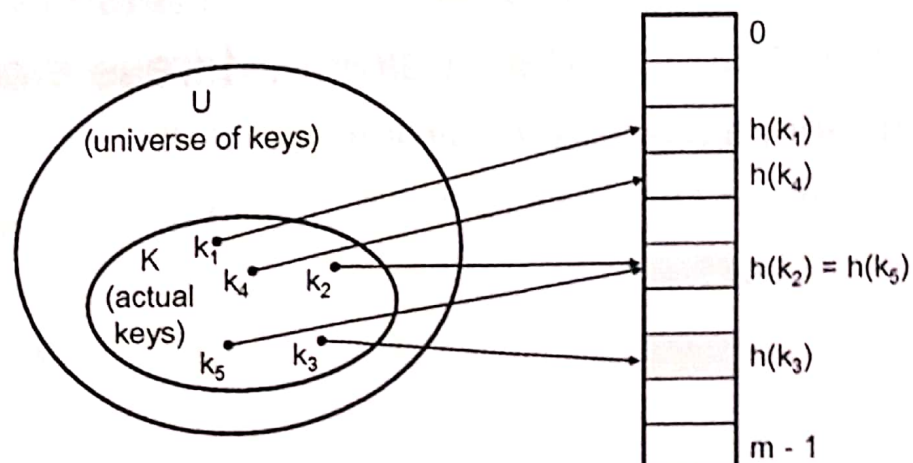
$h : U \rightarrow \{0, 1, \dots, m - 1\}$ We say that k **hashes** to slot $h(k)$

Advantages:

- Reduce the range of array indices handled: m instead of $|U|$
- Storage is also reduced

12

Example: HASH TABLES



Revisit Example 2

Suppose that the keys are nine-digit social security numbers

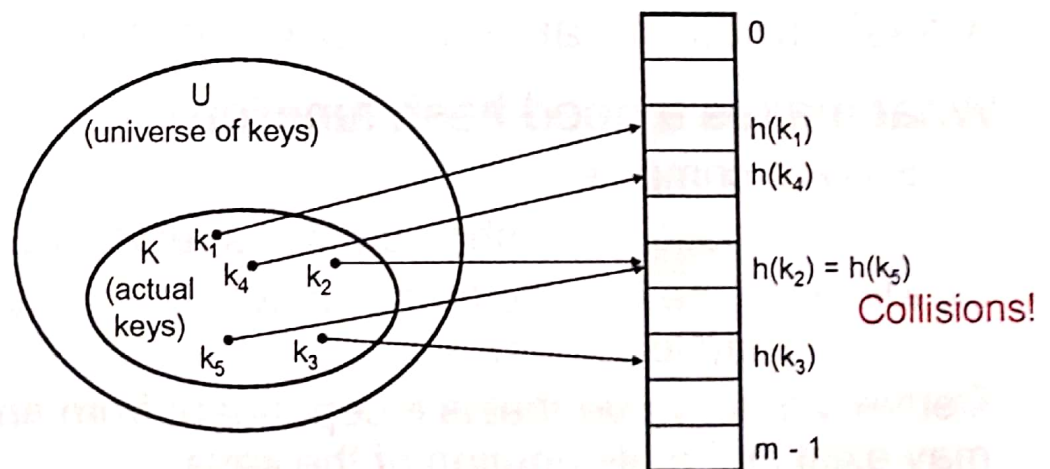
Possible hash function

$$h(ssn) = ssn \bmod 100 \text{ (last 2 digits of ssn)}$$

e.g., if $ssn = 10123411$ then $h(10123411) = 11$



Do you see any problems with this approach?



Collisions

Two or more keys hash to the same slot!!

For a given set K of keys

- If $|K| \leq m$, collisions may or may not happen, depending on the hash function
- If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)

Avoiding collisions completely is hard, even with a good hash function

16

Hash Functions

A hash function transforms a key into a table address

What makes a good hash function?

- (1) Easy to compute
- (2) Deterministic : $h(x)$ should always return the same value
- (3) Minimize the chance that closely related keys hash to the same slot(Collisions)

Derive a hash value that is independent from any patterns that may exist in the distribution of the keys

The Division Method

Idea:

- Map a key k into one of the m slots by taking the remainder of k divided by m

$$h(k) = k \bmod m$$

Advantage:

- fast, requires only one operation

Disadvantage:

- Certain values of m are bad, e.g.,
 - power of 2
 - non-prime numbers

18



Division method

Don't use a power of two. Why?

m	k	$\text{bin}(k)$	$h(k)$
8	25	11001	1
8	1	00001	1
8	17	10001	1

if $h(k) = k \bmod 2^p$, the hash function is just the lower p bits of the value



The Multiplication Method

Idea:

Multiply key k by a constant A , where $0 < A < 1$

Extract the fractional part of kA

Multiply the fractional part by m

Take the floor of the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

Disadvantage: Slower than division method

Advantage: Value of m is not critical, e.g., typically 2^p



Multiplication method

m	k	A	kA	$h(k)$
8	15	0.618	9.27	$\text{floor}(0.27 \cdot 8) = 2$
8	23	0.618	14.214	$\text{floor}(0.214 \cdot 8) = 1$
8	100	0.618	61.8	$\text{floor}(0.8 \cdot 8) = 6$

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

Other hash functions

Cyclic redundancy checks (i.e. disks, cds, dvds)

Checksums (i.e. networking, file transfers)

Cryptographic (i.e. MD5, SHA)

Load factor

The **load factor**, λ , of a hash table is calculated as

$$\lambda = \frac{n}{TableSize}$$

where n is the number of items currently in the table



Handling Collisions

We will review the following methods:

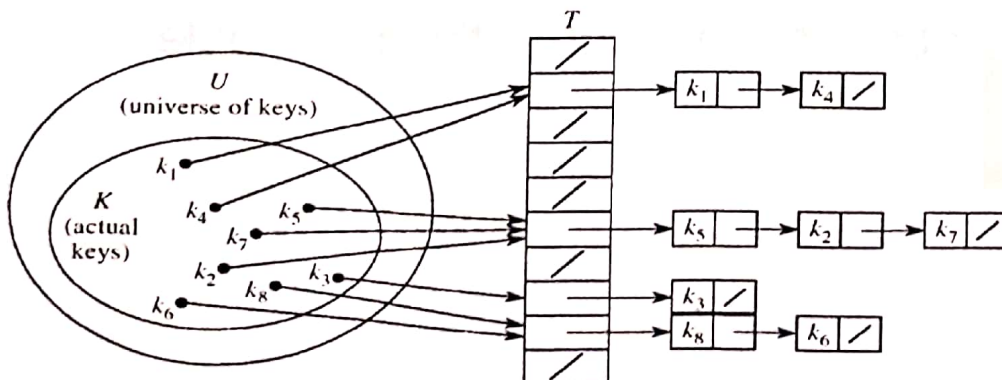
- Separate Chaining
- Open addressing
 - Linear probing
 - Quadratic probing
 - Double hashing

24

Handling Collisions Using Chaining

Idea:

- Put all elements that hash to the same slot into a linked list



Slot j contains a pointer to the head of the list of all elements that hash to j

Collision with Chaining - Discussion

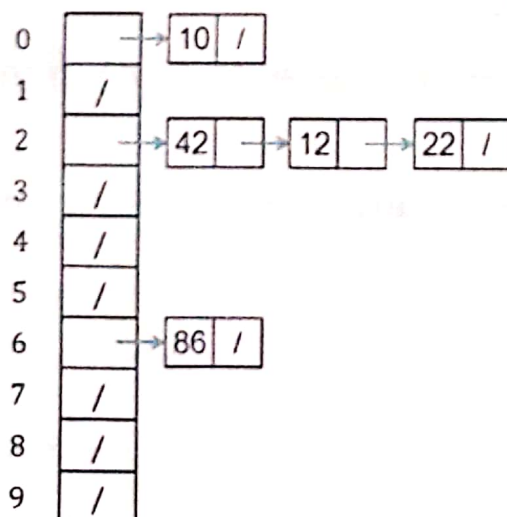
Choosing the size of the table

- Small enough not to waste space
- Large enough such that lists remain short

How should we keep the lists: ordered or not?

- Not ordered!
 - Insert is fast
 - Can easily remove the most recently inserted elements

Separate Chaining



All keys that map to the same table location are kept in a linked list

Example:

insert 10, 22, 86, 12, 42
with $h(x) = x \% 10$



Separate Chaining Analysis

The **load factor**, λ , of a hash table is calculated as

$$\lambda = \frac{n}{TableSize}$$

where n is the number of items currently in the table

Under chaining, the average number of elements per list is λ

So if some inserts are followed by random finds, then on average:

Each unsuccessful find compares against λ items

Each successful find compares against λ items

If λ is low, find and insert likely to be $O(1)$

We like to keep λ around 1 for separate chaining



Separate Chaining Disadvantages

- Parts of the array might never be used.
- As chains get longer, search time increases to $O(n)$ in the worst case.
- Constructing new chain nodes is relatively expensive (still constant time, but the constant is high).

Open Addressing

Separate chaining does not use all the space in the table. Why not use it?

- Store directly in the array cell .
- No linked lists.

Linear probing: Inserting a key

Idea: when there is a collision, check the next available position in the table (i.e., probing)

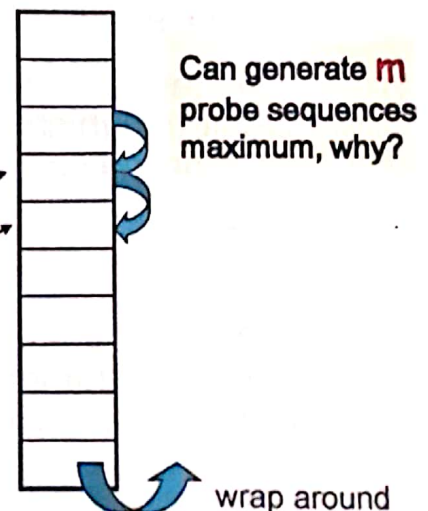
$$h(k,i) = (h_1(k) + i) \bmod m$$
$$i=0,1,2,\dots$$

First slot probed: $h_1(k)$

Second slot probed: $h_1(k) + 1$

Third slot probed: $h_1(k)+2$, and so on

probe sequence: $\langle h_1(k), h_1(k)+1, h_1(k)+2, \dots \rangle$





Open Addressing: Linear Probing

0	8
1	79
2	10
3	
4	
5	
6	
7	
8	38
9	19

How to deal with collisions?

If $h(\text{key})$ is already full,

try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,

try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...

Example: insert 38, 19, 8, 79, 10

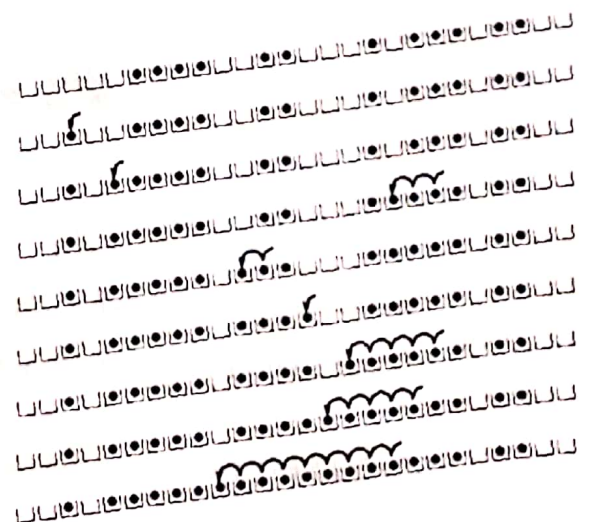
Primary Clustering

It turns out linear probing is a bad idea, even though the probe function is quick to compute (which is a good thing)

This tends to produce clusters, which lead to long probe sequences

This is called *primary clustering*

We saw the start of a cluster in our linear probing example



Linear Probing

Open addressing means resolving collisions by trying a sequence of other positions in the table

Trying the next spot is called probing We just did linear probing
 $(h(\text{key}) + i) \% \text{TableSize}$

Open addressing does poorly with high load factor λ

So we want larger tables

Too many probes means we lose our $O(1)$



Open Addressing: Quadratic Probing

We can avoid primary clustering by changing the probe function from just i to $f(i)$

$$(h(\text{key}) + f(i)) \% \text{TableSize}$$

For quadratic probing, $f(i) = i^2$:

0th probe: $(h(\text{key}) + 0) \% \text{TableSize}$

1st probe: $(h(\text{key}) + 1) \% \text{TableSize}$

2nd probe: $(h(\text{key}) + 4) \% \text{TableSize}$

3rd probe: $(h(\text{key}) + 9) \% \text{TableSize}$

...

i^{th} probe: $(h(\text{key}) + i^2) \% \text{TableSize}$

Probes quickly "leave the neighborhood"



Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

TableSize = 10
insert(89)



Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

TableSize = 10
insert(89)
insert(18)

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

Quadratic Probing Example

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

$49 \% 10 = 9$ collision!

$(49 + 1) \% 10 = 0$

insert(58)



Quadratic Probing Example

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

insert(58)

$58 \% 10 = 8$ collision!

$(58 + 1) \% 10 = 9$ collision!

$(58 + 4) \% 10 = 2$

insert(79)

Quadratic Probing Example

0	49
1	
2	58
3	79
4	
5	
6	
7	
8	18
9	89

TableSize = 10

insert(89)

insert(18)

insert(49)

insert(58)

insert(79)

$79 \% 10 = 9$ collision!

$(79 + 1) \% 10 = 0$ collision!

$(79 + 4) \% 10 = 3$

Another Quadratic Probing Example

		TableSize = 7	
		Insert:	
0		76	(76 % 7 = 6)
1		40	(40 % 7 = 5)
2		48	(48 % 7 = 6)
3		5	(5 % 7 = 5)
4		55	(55 % 7 = 6)
5		47	(47 % 7 = 5)
6			

Another Quadratic Probing Example

		TableSize = 7	
		Insert:	
0		76	(76 % 7 = 6)
1		40	(40 % 7 = 5)
2		48	(48 % 7 = 6)
3		5	(5 % 7 = 5)
4		55	(55 % 7 = 6)
5		47	(47 % 7 = 5)
6	76		





Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	$(76 \% 7 = 6)$
40	$(40 \% 7 = 5)$
48	$(48 \% 7 = 6)$
5	$(5 \% 7 = 5)$
55	$(55 \% 7 = 6)$
47	$(47 \% 7 = 5)$

Another Quadratic Probing Example



0	48
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	$(76 \% 7 = 6)$
40	$(40 \% 7 = 5)$
48	$(48 \% 7 = 6)$
5	$(5 \% 7 = 5)$
55	$(55 \% 7 = 6)$
47	$(47 \% 7 = 5)$

Another Quadratic Probing Example

0	48
1	
2	5
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)



Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76 (76 % 7 = 6)

40 (40 % 7 = 5)

48 (48 % 7 = 6)

5 (5 % 7 = 5)

55 (55 % 7 = 6)

47 (47 % 7 = 5)

(47 + 1) % 7 = 6 collision!

(47 + 4) % 7 = 2 collision!

(47 + 9) % 7 = 0 collision!

(47 + 16) % 7 = 0 collision!

(47 + 25) % 7 = 2

Will we ever get a 1
or 4?!?

From Bad News to Good News

After TableSize quadratic probes, we cycle through the same indices

The good news:

For prime T and $0 \leq i, j \leq T/2$ where $i \neq j$,

$(h(\text{key}) + i^2) \% T \neq (h(\text{key}) + j^2) \% T$

If TableSize is prime and $\lambda < \frac{1}{2}$, quadratic probing will find an empty slot in at most TableSize/2 probes

If you keep $\lambda < \frac{1}{2}$, no need to detect cycles as we just saw

Double Hashing

A second hash function is used to drive the collision resolution